THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

From Software to Hardware: Making Dynamic Multi-core Processors Practical

Paul-Jules Micolet

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2019

Abstract

Heterogeneous processors such as Arm's big.LITTLE have become popular as they offer a choice between performance and energy efficiency. However, the core configurations are fixed at design time which offers a limited amount of adaptation. Dynamic Multi-core Processors (DMPs) bridge the gap between homogeneous and fully reconfigurable systems. They present a new way of improving single-threaded performance by running a thread on groups of cores (compositions) and with the ability of changing the processor topology on the fly, they can better adapt themselves to any task at hand. However, these potential performance improvements are made difficult due to two main challenges: the difficulty of determining a processor configuration that leads to the optimal performance and knowing how to tackle hardware bottlenecks that may impede the performance of composition.

This thesis first demonstrates that ahead-of-time thread and core partitioning used to improve the performance of multi-threaded programs can be automated. This is done by analysing static code features to generate a machine-learning model that determines a processor configuration that leads to good performance for an application. The machine learning model is able to predict a configuration that is within 16% of the performance of the best configuration from the search space.

This is followed by studying how dynamically adapting the size of a composition at runtime can be used to reduce energy consumption whilst maintaining the same speedup as the fastest static core composition. An analysis of how much energy can be saved by adapting the size of the composition at runtime is conducted, showing that dynamic reconfiguration can reduce energy consumption by 42% on average. A model is then built using linear regression which analyses the features of basic blocks being executed to determine if the current composition should be reconfigured; on average it reduces energy consumption by 37%.

Finally the hardware mechanisms that drive core composition are explored. A new fetching mechanism for core composition is proposed, where cores fetch code in a round-robin fashion. The use of value prediction is also motivated, as large core compositions are more susceptible to data-dependencies. This new hardware setup shows massive potential. By exploring a perfect value predictor with perfect branch prediction and the new fetching scheme, the performance of a large core composition can be improved by a factor of up to 3x, and 1.88x on average. Furthermore, this thesis shows that state-of-the-art value prediction with a normal branch predictor still leads to good performance improvements, with an average of 1.33x to up to 2.7x speedup.

Layman

As computers become more and more powerful, the main component that drives this performance — the central processing unit — is becoming harder to improve on due to a multitude of reasons. In order to compensate for the fact processor designers cannot increase the performance of a single processor core; the common philosophy has been to pack multiple copies of smaller cores into a single package. Whilst this approach has been successful to a certain degree, some applications cannot benefit from spreading their computation on multiple processor cores but instead perform best on larger cores. To compensate for this, hardware designers propose heterogeneous processors where different types of cores are on the same package. However, this still requires that the design of the processor be determined before it is built. This reduces the flexibility of the system and thus may potentially limit performance on some applications. Therefore, a new processor model was created, one where the cores can be grouped together at any time in order to form larger cores: these processors are called Dynamic Multi-core Processors (DMPs). DMPs can adapt themselves to the needs of the program at hand which helps improve the performance of the program.

This thesis explores how to determine ways in which the DMP can automatically reconfigure itself to adapt to the needs of the program at hand. Whilst previous research has looked into ways of reconfiguring the processor using hand-crafted algorithms, this is not an efficient solution as it requires intimate knowledge of the processor and is time consuming. Automating this process makes these types of processors more accessible, as programmers will not have to think about the underlying hardware when they write their programs. The reconfiguration can be used to either make the programs execute faster, or consume less energy. This thesis shows that reconfiguring the processor can be fully automated through the use of machine learning, thus removing the need of hand-crafting an algorithm. Finally, the mechanisms that determine how the processor functions once reconfigured are analysed to see if any improvements can be made. By modifying the way the processor behaves, the performance of programs can be improved even more. This motivates further research in these types of processors.

Acknowledgements

Fully documenting how people around me have helped me through this endeavour would require a thesis of its own, but in the interest of time (and space) I will try to be succinct.

- First and foremost I would like to thank my supervisor Christophe Dubach for having guided me throughout these four years. It was certainly not easy, even at the very end, but your advice was instrumental to the completion of this project. I would also like to thank Aaron Smith for having given me the tools to conduct my research.

- Next, I would like to thank the team in room 1.34; Harry Wagstaff, Tom Spink, Stephen Kyle, and (sure, different room) Bruno Bodin. You kept my somewhat short stay at the Informatics Forum always interesting and motivated me to see this project through. Sure, we may have spent more time probably discussing non-research things over bad coffee but it made going to the Forum worth it.

- As for those who had to experience my various rants about my thesis, I hope I don't forget anyone. Gerald Chau, Yordan Petrov, Nick Nikolov, Stoyan Aleksandrov, Ellen Nevrokopska, Gosia Waszak, Nicholas Swafford and David Rankin. In some form or another you have all had to bear with me for the past four years and I thank you all for it.

- I would also like to thank my parents, brothers and family for having believed in me and supported me throughout. Even when things were not very clear, you trusted my ability to get through it.

- Finally, I would especially like to thank Lea Aliperta-Mourissoux. You have been on the front line for the entirety of my PhD, seeing me through the stress of publishing and even coming with me to conferences all the way to Korea. Without your support I know I could not have done it and I will always be grateful for it.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- **Paul-Jules Micolet**, Aaron Smith, and Christophe Dubach. 2016. A machine learning approach to mapping streaming workloads to dynamic multicore processors. In Proceedings of the Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES 2016).
- **Paul-Jules Micolet**, Aaron Smith, and Christophe Dubach. 2017. A Study of Dynamic Phase Adaptation Using a Dynamic Multicore Processor. ACM Transactions on Embedded Computing Systems (TECS 2017).

(*Paul-Jules Micolet*)

Table of Contents

Chapter 1

Introduction

Multi-core processors are now common in all computing systems ranging from mobile devices to data centres. As advances in single-threaded performance have slowed, multi-core processors have offered a way to use the increasing numbers of transistors available. Due to the difficulty of designing more powerful cores, a shift towards adding more cores into a single package seems inevitable. These high core-count processors are a subset of multi-core processors known as tiled architectures. A tiled architecture such as Tilera [Bell 08] or Raw [Wain 97] is composed of smaller simpler cores that are placed on a regular grid. This improves hardware scalability and enables multi-threaded applications to exploit the large core count.

Yet, workloads that require high single-threaded performance are penalised by the simple nature of each core [Eyer 10]. One solution to this problem is heterogeneous multi-cores which utilise cores with different levels of power and performance. Although heterogeneous multi-cores are common place in mobile devices, they have little reconfiguration or adaptive capabilities (e. g. only two type of cores available for Arm big.LITTLE). Dynamic multi-core processors (DMPs) offer a solution to this problem by allowing cores to compose (or fuse) together [Ipek 07] into larger compositions to accelerate single threads. This produces "on-demand" heterogeneity where cores are grouped to adapt to the workload's demand.

Whilst the flexibility of a DMP is an advantage, it increases the complexity of obtaining the optimal performance out of applications. If reconfiguration cannot be automated, then it is up to the programmer to decide how the processor must be partitioned, adding extra development work. In order to motivate further development of DMPs, it is important to prove that they can be practical. This thesis demonstrates that different techniques can improve the usability and performance of DMPs, making them an attractive approach towards processor design.

1.1 The Problem

This section covers the different problems that are considered a limiting factor of DMPs in terms of their practicality.

Reconfiguration Whilst there exists a multitude of proposed dynamic multi-core processor architectures as seen in Mittal's survey [Mitt 16]; work on understanding when to compose cores, or what type of programs can benefit the most out of core-composition is scarce. A 16 core DMP for example has over 32,000 configurations when executing multi-threaded programs, making exhaustive search of the space impractical. Therefore, without some method of automating the reconfiguration of the processor the programmer must have intimate knowledge of both the architecture and the programs that will execute on them in order to determine a good configuration.

Previous work on determining how many cores must be composed for a given program focus on using profiling information [Pric 14] or heuristics based on observations [Gula 08]. They consider core-composition to be a *black box*: instead of trying to understand what features of a program lead to good performance, they evaluate it on different core-composition sizes and determine the best one. The best configuration can be the one that leads to the fastest execution time, or the most efficient one based on a specific metric such as energy/power efficiency. This approach makes DMPs less practical as it increases the amount of work required to ensure that workloads benefit from core-composition.

If a configuration of the processor that fits the user's requirements could automatically be determined without requiring multiple executions of the program at hand, then this would make the process of getting the best performance lightweight. This would allow programmers to modify their programs without having to extensively re-profile their applications, making dynamic multi-core processors more approachable. There are two possible methods of achieving this: either by having the hardware reconfigure itself automatically by analysing the executing program, or by having the compiler determine the configuration via an analysis of the program. Using the hardware allows for greater flexibility as legacy software can immediately benefit from the reconfiguration without having to be re-compiled.

Maximising performance through software optimisations Not all applications benefit from executing on a core-composition, reducing the attractiveness of implementing the feature in a processor. The lack of performance may be due to the fact that the pro-

grams are not designed to be executed on a system that supports core-composition. Programmers may have to re-write their code to ensure that a program that currently does not perform any better could not be re-written to benefit from the new hardware.

However, there exists no information as to what optimisations may help improve the performance of applications on a dynamic multi-core processor. Furthermore, a programmer may not necessarily have access to a compiler that provides passes that are targeted towards such systems. In this case, it is important to explore source-level optimisations and study how they can help increase the performance of programs on core-composition. By underlining which optimisations will lead to performance improvements, this encourages the ease-of-use of dynamic multi-core processors.

Core composition mechanisms Finally, as the concept of dynamic multi-core processors is still relatively new compared to other processor designs, the currently proposed techniques may not be enough to maximise performance. For example, core-composition exploits instruction level parallelism (ILP) by executing a single thread on multiple cores. Unfortunately unresolved data-dependencies, that often arise when many instructions can be executed in parallel, will reduce the potential performance improvements as instructions must wait on each other to execute. This means that DMPs will not improve performance to the fullest of their capacities.

Previous work attempts to resolve this problem by applying register forwarding to try and reduce the latency caused by data-dependencies [Roba 11]. However, this is only a partial solution as instructions must still wait on instructions they are dependent on, reducing the potential improvements. It is therefore crucial to find new ways of addressing the bottlenecks of core-composition from a hardware perspective. By proposing new solutions to these bottlenecks, core-composition can become more effective at improving the performance of programs. Introducing new hardware can even help dynamic multi-core processors become more adaptable to new types of programs, making them more practical.

1.2 Contributions

This thesis tackles the three problems described through the use of different techniques.

Reconfiguration First, this thesis looks at the problem of how to automatically reconfigure the processor using techniques that learn from previous executions of programs. A set of machine-learning models that are able to automatically make config-

uration decisions based on features of the software are proposed. This thesis presents a design methodology for generating these models that use either program features extracted at the source code level or features that can be extracted at runtime, to determine when and how to configure the cores. These models are not influenced by hand-picked heuristics, but instead are generated by exploring how different configurations and programs' features affect the affect performance and feeding this information to a machine-learning model. By using machine learning, the processor can automatically be reconfigured ahead of time to improve the performance of multi-threaded applications, speeding up execution by 2.5x on average compared to a single core and up to 10x in the best case. The thesis demonstrates that the processor can also be reconfigured at runtime to reduce energy consumption by 37% on average whilst maintaining the speedup of the best static configuration.

Maximising performance through software optimisations This thesis also provides an analysis of how different features of the source-code affect the speedup obtained via core-composition. This is achieved by exploring a set of source-level optimisations and studying how this influences the performance of core-composition. The analysis provides insights on when core-composition should be used, and how programs may be modified to increase the potential speedup. It also motivates that the code modifications can be relatively fast to implement; demonstrating that even without access to the source code, programs can be tuned to benefit from DMPs.

Core composition mechanisms Finally, the thesis covers some of the current shortcomings of core-composition from a hardware perspective. The two main features of the hardware that are analysed are how instructions are fetched when cores are composed and how data-dependencies affect the performance of large core-compositions. This thesis underlines that a simple, serial fetching scheme for instructions can be a considerable bottleneck when attempting to populate large core-compositions. It also explores how data-dependencies limit the amount of ILP that large core-compositions can extract. By providing a new fetching scheme that parallelises instruction fetches, and a value predictor which can speculate the results of instructions, this thesis shows that the performance of core-composition can be improved. The thesis shows promising results: the performance (speedup) of core-composition can be improved by 1.87x, and up to 3x in the best case with perfect value and branch prediction; whereas using a state of the art value predictor still leads to an average performance increase of 1.33x and up to 2.7x. This motivates further research into value prediction.

1.3 Structure

The overall aim of this thesis is to provide methods of making DMPs more practical, from automated reconfiguration to new hardware that improves the performance of the DMP. The structure of the thesis is as follows:

Chapter 2 provides information on the different topics used throughout this thesis. The topics involve the reconfiguration mechanisms of DMPs, the instruction set architecture used, how value prediction works and the different machine-learning techniques employed throughout this thesis.

Chapter 3 presents the related work. This covers previously proposed DMP processors and the different offline and online reconfiguration schemes that have been explored. This is followed by a discussion of work done on compiler optimisations, the different hardware techniques that improve energy efficiency, other proposed value predictors and different types of speculative hardware.

Chapter 4 covers the setup of the simulator used throughout this thesis and the benchmarks that are explored.

Chapter 5 shows how the process of configuring a DMP to improve the performance of multi-threaded streaming applications can be learned. The chapter demonstrates that a mix of core-composition and multi-threading is required to get the best speedup for these applications. A machine-learning model is then trained to determine a good configuration of the processor based on source-code information derived from the application. This chapter is based on the work previously published at LCTES 2016 [Mico 16].

Chapter 6 uses dynamic reconfiguration to reduce energy consumption whilst maintaining the same performance as the optimal static ahead of time configuration. The chapter first covers some of the factors that affect the performance of core-compositions, such as branch prediction accuracy requirements. This is followed by a study of how runtime adaptation of the processor can reduce energy consumption. A machine-learning model that can determine the correct size of a core-composition at runtime, based on the types of instruction being executed is then designed. This proves that dynamic reconfiguration can also be learned. The chapter is based on the work previously published at CASES 2017 [Mico 17].

Chapter 7 presents new hardware that allows larger core-compositions to perform better in the average situation by reducing latencies caused by executing code on such compositions. The new additions involve a new fetching mechanism that ensures each

core can fetch instructions independently and in a round-robin fashion and the use of value prediction to minimise stress on the network on chip and reduce the effect of data dependencies between cores. The chapter shows that a current state-of-the art value predictor paired with the new fetching mechanism can outperform the current implementation of core-composition.

Chapter 8 finally concludes this thesis by summarising the contributions, providing critical analysis and presenting future work in the field of DMPs.

Chapter 2

Background

This chapter covers the different topics that are present in this thesis and is structured as followed:

- Sections 2.1 and 2.2 cover Chip Multi-core Processors and Heterogeneous Chip Multi-core Processors to motivate the use of Dynamic Multi-core Processors (DMP).
- Section 2.3 describes the three types of DMP that currently exist.
- Section 2.4 describes the Explicit Data Graph Execution (EDGE) instruction set architecture (ISA) used by the DMP explored throughout this thesis. Subsection 2.4.1 explains how EDGE blocks are formed, which is useful for Chapter 5 and Chapter 6. Subsection 2.4.2 covers how blocks are executed on a single core, as well how they are executed when the cores are composed, this is important knowledge for Chapters 5, 6 and 7.
- Section 2.5 explains how value predictors works, this is used in Chapter 7.
- Section 2.6 describes streaming programming languages, explored in Chapter 5.
- Section 5.4 covers the different machine-learning techniques and the evaluation method used. Linear regression, described in subsection 2.7.1 is used in both Chapter 5 and 6. k-Nearest Neighbours explained in subsection 2.7.3 is used in Chapter 5, k-means clustering found at subsection 2.7.2 is used in Chapter 6. Leave-one cross validation described in subsection 2.7.4 is used in Chapter 5 and 6.
- Finally, Section 2.8 explains the Early Stopping Criterion used in Chapter 5.

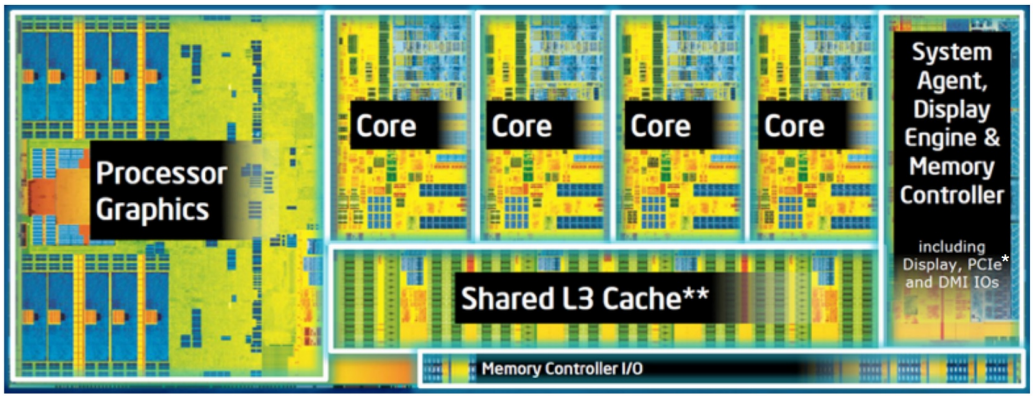


Figure 2.1: Intel Core i7 processor internal die photograph taken from [Turl 14]

2.1 Chip Multi-core Processors

Chip Multi-core Processors (CMPs) are now ubiquitous in the desktop, server and smartphone space due to the difficulty in scaling single-core performance. In a CMP, multiple processor cores are packaged on a single die as seen in Figure 2.1. The most commonly adopted CMP design features homogeneous cores as it reduces the design complexity from a hardware perspective [Asan 06]. In a CMP the performance improvements come from running multiple tasks in parallel. These tasks can either be different programs or multiple threads from a single program executing on multiple cores.

The performance benefits of executing a program on a CMP can be estimated using Amdahl's Law [Amda 67]. It states that the speedup S obtained by executing a program on n cores depends on the fraction of work which is parallelisable f.

$$S = \frac{1}{(1-f) + \frac{f}{n}} \tag{2.1}$$

If a CMP features an infinite number of cores [Eyer 10], then Amdahl's law can be rewritten as:

$$\lim_{n \to \infty} S = \frac{1}{(1-f)} \tag{2.2}$$

Therefore, given any program, the speedup obtained through using a CMP is limited to the fraction f of parallel work found in the program. As all the cores are homogeneous, if the parallel fraction is low, then this causes serial bottlenecks to reduce the potential speedup, as no core will be specialised for single-threaded execution.

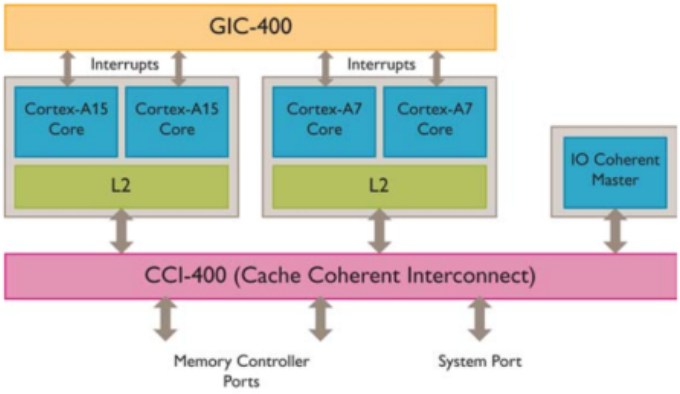


Figure 2.2: Example of a heterogeneous multi-core processor [ARM 13]

2.2 Heterogeneous Chip Multi-core Processors

Unlike CMPs, Heterogeneous Chip Multi-core Processors (HCMPs) bring a variety of cores onto a single package. This variety may come in different forms, such as having multiple instruction set architectures (ISA) on the same system on chip (SoCs) [Venk 14, Venk 16] or different size cores on an SoC [ARM 13, Jeff 12]. Figure 2.2 shows a schemata for Arm's big.LITTLE HCMP, where a high-performance Cortex-A15 is paired with a simpler, power efficient Cortex-A7. The two cores are connected via a cache-coherent interconnect which provides data coherence at the bus-level, allowing the cores to make reads to its neighbour [ARM 13]. Software is then executed on one of the cores depending on the user's requirement.

Unlike CMPs, the variety of cores provide a flexibility to the hardware. This can be used for different purposes, such as security [Venk 16], energy/power savings and speeding up applications [Venk 14]. Whilst the hardware diversity in HCMPs is an advantage compared to CMPs, it increases the complexity of the optimisation space. For example, Gupta *et al.* [Gupt 17] show that a single-ISA octa-core big.LITTLE architecture can have 20 different CPU core configurations, combined with the ability to dynamically modify the voltage, this leads to 4000 unique possible hardware configurations to choose from at runtime. This highly increases the complexity of obtaining the correct settings for different programs. Multi-ISA HCMPs also face a similar issue as having more than a single ISA not only adds design challenges, but program migration between different cores may in fact deteriorate performance [DeVu 12].

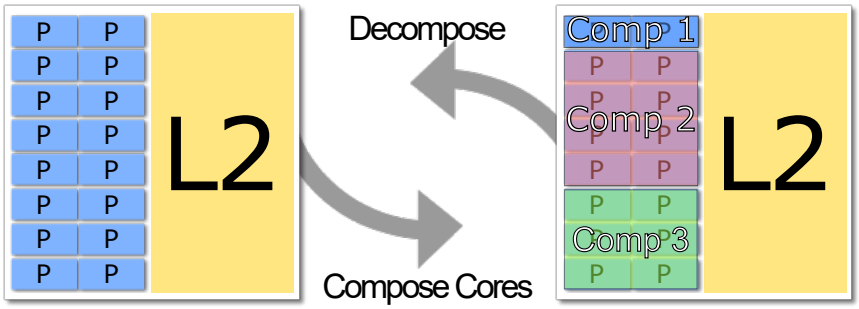


Figure 2.3: High-level view of a DMP that can modify its core count.

2.3 Dynamic Multi-core Processors

In both CMPs and HCMPs, the design of the processor is fixed, meaning that many of the trade-offs between power, performance and area cannot be changed after production. Dynamic Multi-core Processors (DMPs) attempt to bridge the gap between the two previous designs by allowing the execution substrate to adapt dynamically at runtime. Mitall's survey [Mitt 16] defines three types of modifiable resources: the core count [Ipek 07, Kim 07, Pric 12], number of resources that each core has [Homa 12] and micro-architectural features [Fall 14, Baue 08, Tava 15].

Core Composition Dynamic Multi-core Processors A DMP that can modify its core count is built out of homogeneous cores with a reconfigurable fabric on top. Physical cores can function either on their own or as a group. Figure 2.3 provides an high level view of a DMP: the physical cores *P*, are composed into 3 separate compositions of sizes 2, 8 and 4. Each of the compositions executes a single thread, leveraging the power of the cores that are grouped together. A composition fetches instructions from a single source and executes them across all the physical cores that compose it. Cores can fuse dynamically and create core compositions of any sizes. The exact mechanism of core composition is described later on in Section 2.4.

The advantage of a core composition DMP over the traditional CMP or HCMP is the ability to reconfigure the processor dynamically to better match the tasks at hand. For example, large sequential sections of code with high Instruction-Level Parallelism (ILP) can be accelerated on a core composition that mimics a wide superscalar processor. On parallel workloads the DMP can be reconfigured by de-composing the composition as seen in Figure 2.3 to match the Thread-Level Parallelism (TLP).

Resource Sharing Dynamic Multi-core Processors A more fine-grained DMP technique is resource-sharing. For example in the WiDGET DMP by Watanabe *et al.* [Wata 10], cores are built out of Instruction Engine front-ends which function similarly to Out of Order (OoO) cores' front and back end pipeline functions. They then are connected to Execution Units which they can choose to use. Each core in WiDGET also has access to their neighbour's Execution Units, allowing for more variation. Another example of resource sharing is Rodrigues *et al.*'s work [Rodr 14] where a core can use resources such as Arithmetic Logic Units (ALUs) from other cores.

Micro-architectural Reconfigurable Dynamic Multi-core Processors A final example is a DMP which can reconfigure micro-architectural features to better fit the current application. Fallin *et al.* [Fall 14] observe that serial code can exhibit phases that fit different micro-architectural features. According to them, these phases may only been in the ten to hundred thousands instructions long. These DMPs can therefore modify micro-architecural features, such as in-order or out-of-order execution, to best match the current phase of a program.

2.4 EDGE

This section describes the Explicit Data Graph (EDGE) architecture which is the architecture used in this thesis. First the ISA is covered, with an example of how source code is transformed into blocks of instructions. Then the features of an EDGE processor are described, such as the ability of executing multiple blocks on a core and how core composition works with EDGE.

2.4.1 EDGE Instruction Set Architecture

The Explicit Data Graph Execution (EDGE) architecture [Burg 04] is a data-flow architecture aimed at improving concurrency whilst being energy and power efficient [Smit 06a, Burg 04]. Similar to very long instruction word (VLIW) architectures that pack multiple sub-instructions into a single instruction, EDGE requires that the compiler pack instructions into blocks. Unlike VLIW that uses static placement and static issue – which puts high pressure on the compiler – EDGE allows for dynamic issue which distributes responsibility between the hardware and the compiler more evenly.

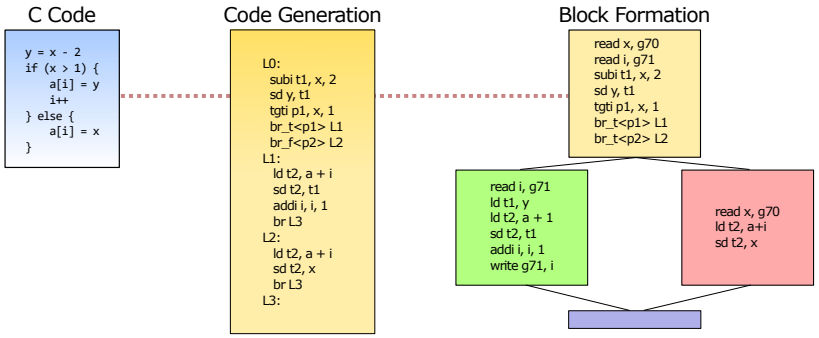


Figure 2.4: High-level view of the two main EDGE compiler passes. First pass involves standard optimisations, and code generation. Second pass transforms the code into atomic blocks of instructions [Smit 06a].

Block Formation In EDGE, instructions are organised into blocks, which are fetched as single units by the processor. Figure 2.4 shows a high-level overview of how EDGE creates the blocks of instructions; this is a summary of the work by Smith *et al.* [Smit 06a]. The first pass (Code Generation) transforms source code into a control flow graph (CFG), performing optimisations such as loop unrolling and inlining. Then, each node of the CFG is turned into an EDGE block given a set of restrictions. These restrictions are:

- Block Size: an EDGE block may be between 4 to 128 instructions.
- Load/Store: an EDGE block may have at most 32 load/store instructions.
- Entry/Exit: an EDGE block may have a single entry point which is the first instruction in the block but may have up to 3 exits (two predicated branches and a fall-through branch).

If a block does not meet these requirements, it is broken down into smaller blocks.

Unlike traditional ISAs such as x86 or Arm, instructions in a block do not communicate via registers, but rather the output targets of instructions are encoded to instruction inputs [Smit 06a]. Loads and stores in each EDGE block are assigned unique identifiers which are used to resolve load-store dependencies. Thus, the EDGE architecture encodes dependencies between instructions at the ISA level. Registers are also only used for inter-communication between blocks.

An EDGE block contains a header that will inform the hardware about the number of stores and register writes contained in the block. This information is used to detect data-dependencies between blocks if multiple blocks are executed in parallel. For example, if two blocks write to the same register, then the register-file can detect which one must fire the write first based on the header information.

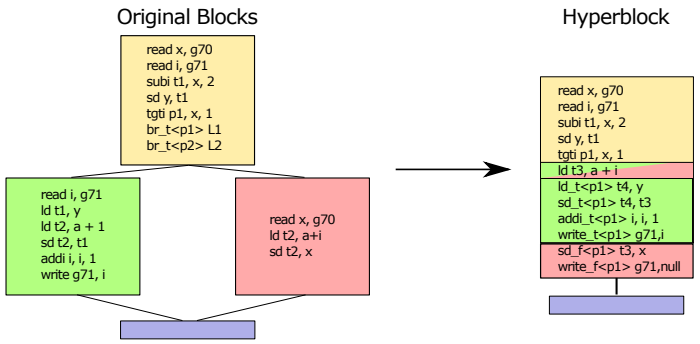


Figure 2.5: High-level view of hyperblock formation. The three top blocks can be fused into a single block using instruction predication.

Hyperblock formation To increase the size of EDGE blocks, multiple blocks can be combined together to form one large block called a hyperblock. This is achieved through the use of instruction predication [Smit 06a]. For example, the *if/else* statement in the C code found in Figure 2.4 normally generates two blocks, one for each statement. However, using predication, the two blocks can be fused together and appended to the block that precedes the *if/else*, reducing the number of blocks from 3 to 1. Figure 2.5 shows the organisation of the new block, where the colour represent where the instructions originated.

As the compiler needs to declare the number of stores and which registers are written to in the block header, extra instructions may need to be generated to ensure the block always executes the same amount of stores and register writes. In the case of Figure 2.5, the last instruction is generated to ensure that a write to the register *g71* always happens regardless of the predicate. This adds some complexity to hyperblock formation, as the size of the block may be artificially inflated if the merging blocks write to different registers or have a different number of stores. Even so hyperblocks allow for increased instruction-level parallelism (ILP) as which in turn improves the performance of the application being executed [Smit 06a]. Overall, the EDGE ISA enables the architecture to dispatch blocks speculatively, with low overhead [Putn 11, Kim 07], therefore, increasing exploitation of ILP.

2.4.2 EDGE Processor features

This section covers two important features of an implementation of EDGE processor called E2 [Putn 11] which is used throughout the thesis. These are the segmentation of the instruction window, and the core-composition mechanism used in later chapters.

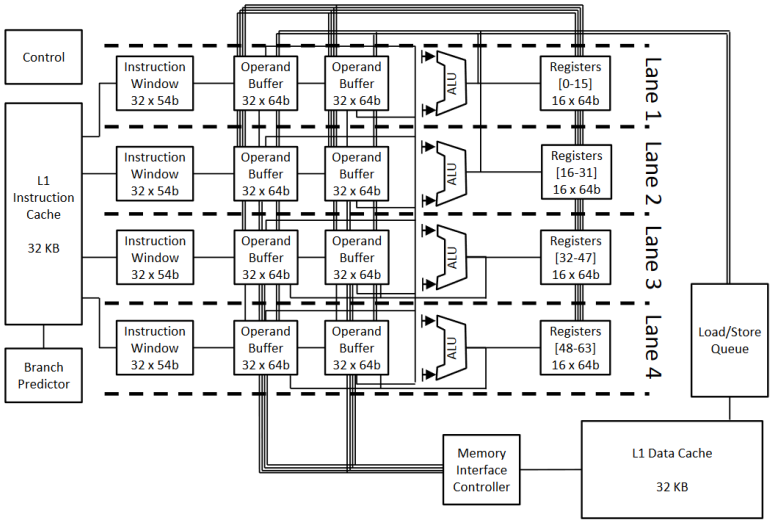


Figure 2.6: Example of a 4 lane core on an EDGE processor taken from [Putn 11]. The instruction window is split into 4 lanes, each can fetch a block of up to 32 instructions.

2.4.2.1 Core Lanes

According to Smith *et al.* [Smit 06a], the sizes of blocks generated by the compiler for a set of applications from SPEC2000 [Henn 00] and EEMBC [Poov 09] range from 5 instructions to 50 instructions and on average are 20.6 instructions long. If an EDGE processor is designed to fetch large blocks, which can be up to 128 instructions long, then the compiler is currently not able to generate large enough blocks. To maximise core-utilisation, an EDGE processor instruction window can be segmented into lanes. This allows the core to fetch more than a single block at a time.

Figure 2.6 shows a schemata for a four-lane EDGE core. Given that a block in EDGE can be up to 128 instructions long, each lane is allowed to fetch a block of up to 32 instructions. Fetching blocks larger than 32 instructions results in more than one lane being occupied by a single block. Depending on the architectural setup, lanes may either share ALUs or they may have private ALUs. As the instruction window is segmented, a core can continue fetching blocks until all its lanes are filled, allowing the processor to have multiple blocks in flight per core. A segmented instruction window therefore allows EDGE cores to be more flexible to block size variability.

2.4.2.2 Core Composition for EDGE

Hardware implementation Core Composition is achieved by fusing a set of *physical* cores to create a larger core composition. In the processor used throughout the thesis this does not modify the physical structure of the processor, instead it provides a unified

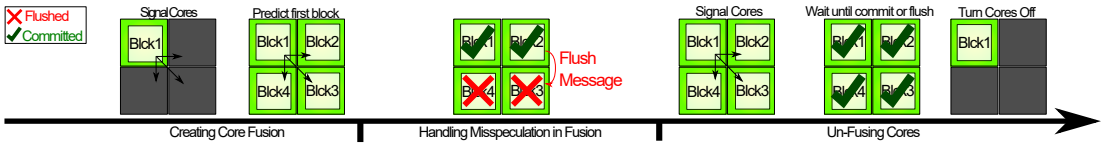


Figure 2.7: Core Composition Mechanisms for the EDGE-based architecture.

view of a group of physical cores to the software. The micro-architecture is distributed: register files, Load Store Queues (LSQs), L1 caches and ALUs all look like nodes on a network. This means that when cores are composed together, this is similar to adding an extra node to the network. Core-composition is a dynamic modification and may occur during the execution of a program to better fit the workload. Unlike traditional CMPs, composed cores operate on the same thread and extract Instruction-Level Parallelism (ILP) rather than Thread-Level Parallelism (TLP) [Mico 16, Pric 12].

Core-composition mechanism Figure 2.7 shows the different stages and mechanisms of core composition for a four core EDGE system. When creating a new composition a master core informs all other cores about the fusion and sends the predicted next block address to an available fused core. When a new thread is started on a fused core the OS and runtime write the new core mapping to a system register. This system register informs the master core which cores belong to the same composition; each bit representing a specific core on the processor. As the register files and caches are distributed when cores are composed, the register is also used to determine which core is responsible for each register or cache line. The hardware then flushes these cores if they are not idle and sets the PC of the first block of that thread on one core in the composition and starts executing.

The topology of the composition can either be hand-picked by a programmer via an API, or simply defined by specifying the number of cores that they wish to have composed. In the first case, the programmer specifies which cores should be fused and calls a function which will trigger the composition; the core that executes this call becomes the master core. In the second case, the programmer only defines the number of cores that must belong to the composition; then when the master core executes the composition call, it chooses the X cores that follow it in the processor mapping.

When a core mispredicts a branch in a composition, it informs the other cores which flush any younger blocks. When un-fusing, the master core informs the other cores, which then commit or flush their blocks and power down while the master core continues to fetch and execute blocks from the thread. The extra hardware required to

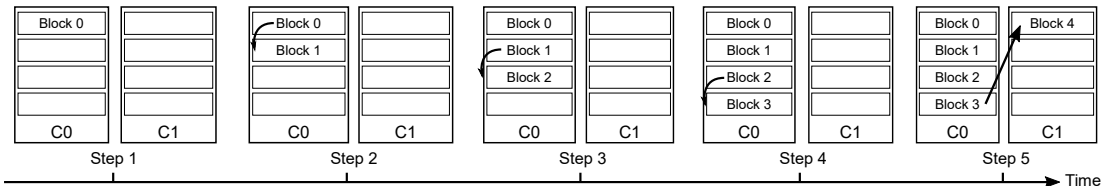


Figure 2.8: Example of the fetching model on a 2 core composition. Each core has 4 segments, the arrows represent the block generating the predictions.

support dynamic reconfiguration is very minimal [Kim 07] since most of the machinery already in place can be reused such as the cache-coherence protocol when fusing and un-fusing the cores.

Fetching on a composition Cores in a composition do not fetch blocks independently; instead a core will only start fetching if it is told to do so by another core in a composition. Figure 2.8 illustrates the fetching mechanism on a 2 core composition where each core has its instruction window segmented into four lanes. The first core, *C0* initiates the composition and starts fetching blocks until it is full. Once this happens, it will send a fetch request to *C1*, which then, and only then, will start fetching blocks. Core *C0* stops fetching blocks after this request, and will have to wait on *C1* to send it a request before it can restart fetching.

Memory and register consistency When a core composition has multiple blocks on different cores, it can execute the instructions out of order. However instructions that modify memory or registers pass through the load-store queue (LSQ) and register-file respectively and are executed in order to ensure memory and register-value consistency. Register reads from younger blocks that depend on a register write from an older block will be stalled until the write is executed. In case of a memory violation caused by undetected dependencies a flush of all blocks younger than the violator, including the violating block, is performed.

2.5 Value Prediction

This section covers the principles of value prediction, which is used in Chapter 7 to resolve data-dependencies at runtime. The different types of predictors are described, and then, the value predictor used in the chapter is explained in detail.

In superscalar processors, instructions can be executed out of order (OoO) to im-

```
1  for(int i =0; i < max_num;i++)
2    b[i] = i;
3  for(int i =0 ; i < max_num ; i++)
4    a[i] = b[i];
```

```
1  loop2:
2  cmp r1, max_num
3  beq exit
4  lsli r7, r1, 2
5  add r3, r2, r7
6  add r5, r4, r7
7  load r6, r3
8  store r6, r5
9  addi r1, r1, 1
10 br loop2
```

Listing 2.1: Small C example, with the second loop's assembly in the right pane

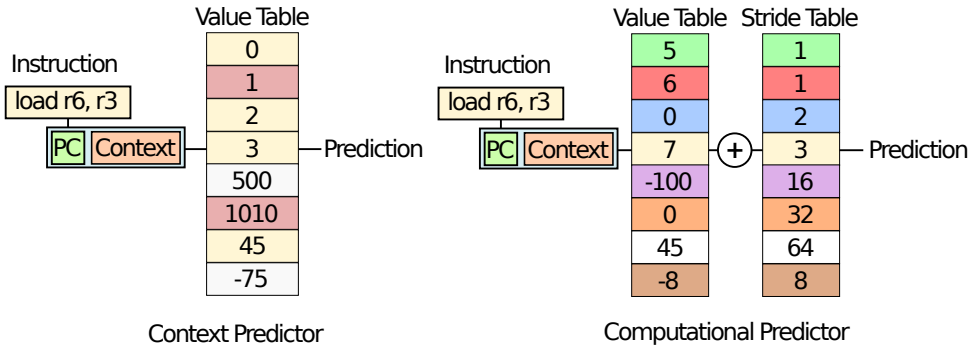


Figure 2.9: Depiction of how predictions are made in a context and computational value predictor. The colours of the cells represent values for different instructions.

prove ILP. However, some instructions may depend on results from previous instructions, or the memory system, such as an L1 cache response. In such a case, the instruction will be stalled until the data is available. These data dependencies reduce the potential ILP, which in turn limits performance.

Value Prediction aims to provide a hardware solution to reducing latencies caused by memory dependencies in OoO superscalar processors [Gabb 98] by predicting the values produced by instructions. Instructions can be speculatively executed with predicted values as they wait for the real values to become available. This is similar to branch prediction, where instructions from a predicted branch path are executed before the branch is resolved. As long as the accuracy of the prediction is high, then executing the data-dependent instructions with speculative data increases ILP.

Type of predictors There exist two categories of value predictors: *Context* predictors and *Computational* predictors. To illustrate the difference between the two, Figure 2.9 depicts how both generate a prediction for the load instruction at line 17 in Listing 2.1. In Figure 2.9, the colour of an entry in the tables represents a value for a specific instruction based on its program counter (PC). The context box represents some information that is used by the predictor, such as global history, or dataflow history.

A *Context* predictor relies on the history of values an instruction outputs to generate a prediction. This means that a single instruction can occupy multiple entries of the predictor as each new value must be recorded. This is why, in Figure 2.9, the specific load instruction has multiple entries in the predictor, as the value generated by the load is different for each iteration of the loop. The *Context* predictor can only predict the value of the instruction if the specific value is already stored in the predictor as it does not have any ability to *infer* the value. The predicted value depends on the context, which can either be captured by dataflow information or branch history [Pera 14]. Overall, whilst predictions may be simpler to generate they are considered space-inefficient [Pera 15].

On the other hand *Computational* predictors make a prediction by computing the value. This can be achieved by determining the delta between two successive values for an instruction, also known as the *stride*. Instead of storing each observed value for an instruction, a *Computational* predictor stores the *stride*, and the last committed value of an instruction. When a prediction is requested, the *Computational* predictor fetches the last committed value and the stride and adds them together to produce the prediction. For example, the value of the array in listing 2.1 is always equal to the previous load value + 1. This means that the *stride* for that instruction is 1 as seen in Figure 2.9. If the last seen value for that instruction was 5, then the next prediction will fetch the value 5 from the last value table, and then add the stride (1), to generate the prediction. This allows *Computational* to potentially reduce the size of the predictor as it only requires a single copy of the value to make a prediction.

2.5.1 The Differential Value Tagged Geometric length Predictor

This section covers a specific implementation of a value predictor that is used in Chapter 7. The Differential Value Tagged Geometric length Predictor (D-VTAGE) [Pera 15] is a Computational value predictor that operates on blocks of instructions to minimise network pressure and increase the prediction output per cycle. To simplify the explanation of the predictor, the size of the block is set to 1 instruction. D-VTAGE is organised in two parts, a Last Value Table (LVT) that contains the last committed values for a given set of instructions, and a set of value tables (VT) which contain the strides. The left-hand side of Figure 2.10 shows an overview of the predictor with N stride tables. The largest stride table is a tagless base predictor that is directly indexed, whereas VT1 to VTN are tagged by hashing the Program Counter of the instruction with varying amounts of the branch history.

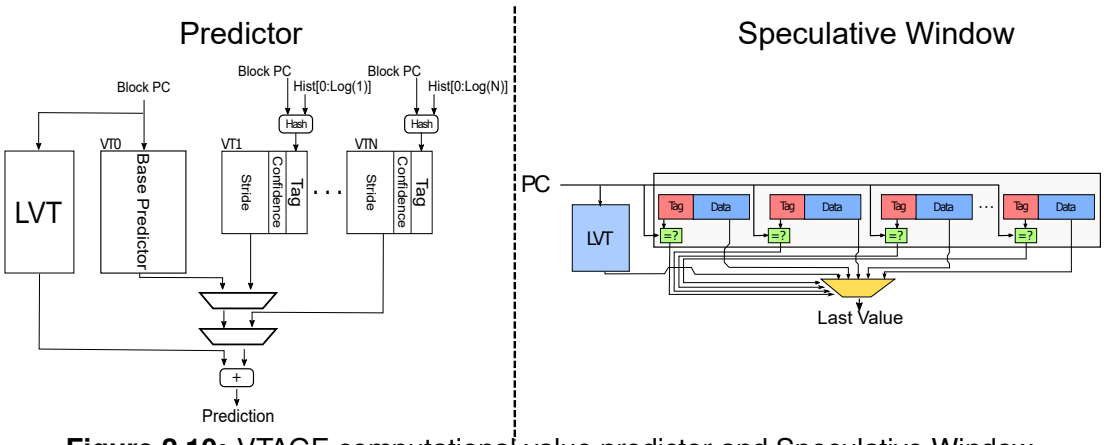


Figure 2.10: VTAGE computational value predictor and Speculative Window.

Confidence Counter A predicted value does not have to be used immediately, as the instruction can still be dispatched normally. This means that the value predictor can be trained before a prediction is used. In D-VTAGE, each entry is paired with a 3-bit saturating counter that is either incremented on a correct prediction, or set to 0 on a misprediction. When a prediction is made, it is only used when the confidence counter is saturated to ensure high accuracy. However, a 3-bit saturated counter may not be enough to ensure that all used predictions are correct.

A method to increase accuracy without increasing the physical size of the counter is to use Forward Probabilistic Counters (FPC) [N 06]. FPCs are essentially a compression technique for counters; instead of incrementing the counter by 1 each time a prediction is correct, it is incremented based on a probability. For example, given a 3-bit counter and a probability vector $\{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$, then the counter has a 100% chance of being incremented from 0 to 1, whilst it only has a $\frac{1}{16}$ chance of incrementing from 1 to 2. This effectively increases the value at which predictions will be used without physically increasing the size of the counter.

Making a prediction All the tables of the value predictor are accessed in parallel using the Program Counter hashed with different amounts of the branch history. The prediction will fetch values from the LVT and the stride tables. The base predictor and LVT are directly indexed, whereas the tables VT1 to VTN are tagged. The stride information will come from the table that has the longest branch history. The prediction is then generated by adding the stride to the value in the LVT. If the confidence counter of the entry is equal to the firing value then the prediction is used, it not then the prediction is kept alive until the instruction is committed to train the predictor.

Updating the predictor Predictions are validated at the commit phase of the instruction to simplify the hardware [Pera 14]. On a correct prediction, the entry that generated the prediction has its confidence counter incremented by one. If the predictor mispredicts, the entry's confidence counter is reset to 0 and the entry is propagated to tables that use longer branch histories. If the entries are propagated to tables with tags, then the PC and branch prediction history are hashed together to form the tag. The prediction and update schemes are directly inherited from the Tagged Geometric (TAGE) branch predictor [Sezn 11].

2.5.1.1 Speculative Window

As predictions are updated at commit time, if multiple blocks with the same PC are live, this can potentially cause multiple value mispredictions as the information in the value predictor may be stale. To ensure this is not the case, current live predictions must be held in a speculative window so that new predictions may use the last predicted values. The right hand side of figure 2.10 demonstrates how the speculative window works. Instead of directly querying the value predictor, the last value table and speculative window are searched in parallel for a matching tag which in this case is the Program Counter (PC) of the block. If there is a live prediction, then the value from the speculative window is used instead of the predictor.

2.6 Streaming Programming Languages

This section covers streaming programming languages, one of which is used in Chapter 5. The features of the language used in that chapter are also described here.

Streaming programming languages are a branch of dataflow programming that focus on applications that deal with a constant stream of data. These applications, such as audio or video decoding can be commonly found in mobile devices. Unlike conventional programming languages such as C++, these languages abstract the concept of incoming and outgoing data to permit the programmer to focus on how the data should be treated. Programs are described as directed graphs where nodes are functions and their edges represent their input and output streams. These languages offer primitives to describe such a graph [Thie 02] which expose parallelisable and serial sections of the application directly to the compiler. Rates of incoming and out-coming data can also be defined to facilitate load-balancing optimisations [Chen 05].

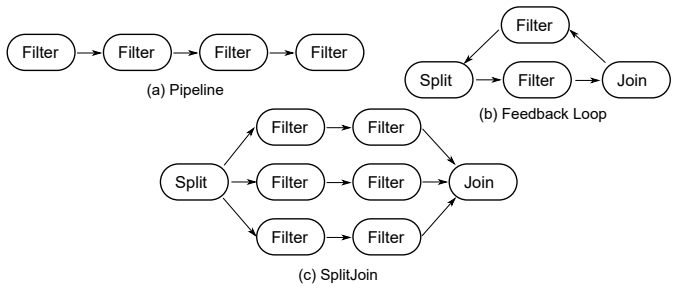


Figure 2.11: Visual representation of the three different StreamIt structures.

Features of streaming programming languages make them an ideal language for targeting multi-core processors. The explicit data communication between the different tasks in the program, the ability to estimate the amount of work performed in each task and information about data rates between tasks allows the compiler to easily generate a multi-threaded application that can run on a dynamic multi-core processor. However, the main challenge consists of deciding how to map the different tasks onto threads and how to allocate the right amount of resources to maximise performance.

2.6.1 StreamIt Programming Language

StreamIt [Thie 02] is a high-level synchronous dataflow streaming programming language that defines programs as directed graphs. StreamIt offers an elegant way of describing streaming applications, abstracting away how infinite data streams are managed to allow the programmer to solely focus on how the data must be treated. A StreamIt program is composed of functions - called *Filters* - which operate on streams of data. Filters declare a certain amount of data which is to be consumed and produced per schedule. Filters can be connected via *Pipelines*, *SplitJoins* or *Feedback Loops* to create the streaming application.

Figure 2.11 displays the different methods of connections in detail. Pipelines (Figure 2.11(a)) represent a sequence of connecting filters operating on the same stream, each filter in the stream will operate on the output of the previous filter. In a SplitJoin (Figure 2.11(c)), data from the stream is passed through a split filter and is either duplicated and passed on in parallel to the filters or distributed amongst the filters in a round-robin fashion. The output of all the filters in a SplitJoin are then concatenated in a round-robin fashion through a joiner filter. Finally a Feedback Loop (Figure 2.11(b)) provides a way for filters to operate on their outputs. The resulting program written in StreamIt represents a graph where the nodes are filters and their edges represent the incoming and outgoing data streams.

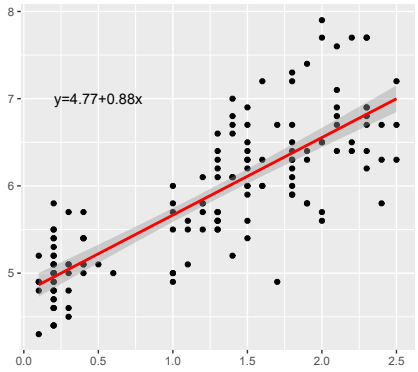


Figure 2.12: Example of linear regression. The inputs are represented by the circles, and the generated linear regression is represented by the red line.

2.7 Machine-learning techniques and evaluation

This section covers the different machine-learning techniques and evaluation methodology used throughout the thesis. The three techniques discussed are: linear regression, k-Means Clustering and k-Nearest Neighbours, whilst the evaluation technique is "leave-one out" cross validation.

2.7.1 Linear Regression

Linear Regression assumes a linear relationship between a set of inputs and the predicted output and generates a model in the form of:

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n \tag{2.3}$$

where y is the predicted output, $X_{1..n}$ are the inputs and $\beta_{0.n}$ are weighted regression coefficients. When training a linear regression, it generates the β weights as to minimise the square-error between the training inputs and predicted outputs. Once a model has been generated from input data, it can be implemented as a set of sums in hardware, making it an efficient way of making predictions. Figure 2.12 shows an example of a linear regression on a small dataset. The linear regression takes the form of $y = 4.77 + 0.88 \times x$ and is represented by the red line.

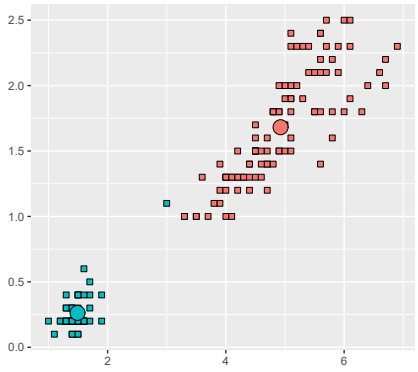


Figure 2.13: Small example of k-Means clustering with two clusters. The squares represent datapoints whilst the circles represent the centroids of the clusters.

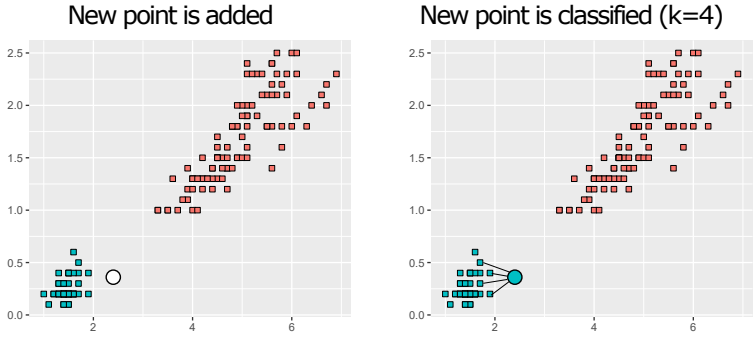


Figure 2.14: Small example of k-Nearest Neighbours classification on a toy dataset with two clusters and k = 4. The squares represent datapoints of the trained kNN dataset. The figure shows a new point (circle) being classified.

2.7.2 k-Means Clustering

k-Means is a clustering technique that groups n observations into k clusters such that $k < n$. The original algorithm presented by Lloyd [Lloy 82] first randomly places k centroids c at random locations in the space. Then, each point x_i in the space of n observations is assigned to a cluster whose centroid c_j is closest to it. Once each point has been assigned to a cluster, each centroid is recomputed by re-centering itself around the datapoints in the cluster. This is repeated until the centroids no longer move.

Figure 2.13 provides a visual example of a running k-Means on a toy dataset, with k set to two. Each colour in the graph represents a cluster, the squares represent the n observations, whilst the circles represent the centroids.

2.7.3 k-Nearest Neighbours

k-Nearest Neighbours (kNN) can be used for both regression and classification. In

both situations an output is generated by averaging the k nearest neighbours to the input from the training data. The average is often obtained by a weighted sum of the features of the k neighbours. Figure 2.14 shows an example of a kNN classification on a new point. The squares represent the trained kNN dataset, the colours represent the different classifications. When a new point is added (as seen on the right hand graph), the four closest neighbours are found, and it is classified under the same category.

2.7.4 Leave-One-Out-Cross-Validation

The process of leave-one-out-cross-validation involves using the training dataset as the evaluation dataset. Given a set of n points in the training set, a new model is trained using $n-1$ of the points available. The last unused datapoint is evaluated against this new model. This procedure is repeated for every point in the dataset, and the results of the evaluation are averaged out to produce a final result. This validation model works well when the number of testing and training data is small.

2.8 Early Stopping Criterion

This section covers the Early Stopping Criterion (ESC) used in Chapter 5 to determine whether or not a sub-sample of the design space is representative of the total space. When exhaustive search of an optimisation space is not feasible, a subset of the space is explored. ESC [Vudu 03] provides a method of determining when new points added in the subspace no longer provide new information, and thus the currently evaluated subspace is representative of the total space. Given N possible number of implementations, if i is a single implementation and x_i is its performance normalised between 0 and 1, then $S = \{x_1, ..., x_N\}$ is the set of all performances for all possible implementations. If X is a random variable corresponding to a randomly selected performance from the set S then there is a cumulative distribution function (CDF) $F(x) = Pr[X \leq x]$ which is equal to the number of items in S that are less than or equal to x (the function $n(x)$) over all possible items N. If, for a number of time steps, a performance from S is randomly chosen and not replaced, then at time step t M_t is maximum performance observed $max_{1 \leq i \leq t} Xi$ up to time t. ESC posits that, that if the probability that M_t is within a distance of the maximum performance found in S then the space exploration can stop at time t. Formally, ESC is:

$$Pr[M_t \leq 1 - \varepsilon] < \alpha \tag{2.4}$$

Where ε is the proximity to the best performance in S and α is the uncertainty factor. For example, if the user wants to stop the search when M_t is within 5% of the best, with an uncertainty factor of 10% they would set ε to 0,05 and α to 0,1. As ESC is used when the exhaustive space has not been searched, the set of all performances S is unknown. ESC thus uses an approximation of the space, using the set of already observed performances rather than S, defined by:

$$\hat{G}_t(x) = \frac{\binom{\lceil N \times \hat{F}_t(x) \rceil}{t}}{\binom{N}{t}} \tag{2.5}$$

Where $\hat{F}_t(x)$ is the CDF for the random variable X given the set of observed performances at time t. Finally, using the approximation, ESC can be defined as:

$$\hat{G}_t(1-\varepsilon) < \alpha \tag{2.6}$$

Chapter 3

Related Work

Whilst dynamic multi-core processors (DMPs) are relatively new in terms of processor design they are still a solution to a well researched set of problems. These problems include improving performance of single-threaded applications and reducing energy consumption. This chapter covers the related work relevant and adjacent to this thesis.

3.1 Reconfigurable Processors

This section covers work on both dynamic multi-core processors (DMP) and processors that can reconfigure their micro-architectures.

3.1.1 Dynamic Multi-core Processors

The idea of composing physical cores was first introduced by Ipek *et al.* in CoreFusion [Ipek 07]. CoreFusion employs a traditional architecture with 2 issue out of order cores. When cores are fused, they collectively fetch from the same thread, whenever an instruction cache miss is issued, an eight word block is delivered and distributed across in all of the cores' caches. Fetches are aligned with the core responsible for the 2 oldest instructions. In the original paper, a 4 core composition obtains a 1.3x speedup on SPEC 2000 INT and a 1.5x speedup on SPEC FP over a 2 issue core.

The Bahurupi [Pric 12] polymorphic heterogeneous multi-core architecture proposed by Pricopi *et al.* introduces a sentinel instruction which informs the hardware about the *live-in* and *live-out* registers of a basic block. Baharupi uses the SimpleScalar Portable ISA (PISA) [Burg 97], and the compiler adds this instruction to the top of every basic block, splitting the program up into blocks similarly to EDGE. They show that, on average this introduces a 24% code size increase for SPEC 2006 INT, 15% for SPEC FP and 19% for Mediabench [Lee 97] and MiBench [Guth 01] benchmarks.

Another important piece of information contained in this instruction is the size of a block. Cores in a composition fetch basic blocks in a similar fashion to the one described in Chapter 2 Section 2.4.2.2. However, cores must execute global register renaming in the sequential order of blocks. To ensure sequential renaming, a Global Program Counter (GPC) is introduced, and each core in the composition must lock the GPC before fetching a block and doing the renaming. Using a 4 core composition they report a performance improvement of 2x on SPEC 2006 INT, 3x on SPEC FP and 4x on Mibench/Mediabench compared to a 2 issue core.

TFlex [Kim 07] proposed by Kim *et al.* is an EDGE processor that also deploys core composition. They motivate that EDGE simplifies distributing instructions across cores as EDGE itself is designed for distributed micro-architectures. Since instruction dependency orders are determined at compile time and encoded at the ISA level, the TFlex instruction fetching can be decentralised, as there is no need for centralised analysis like in a traditional ISA (such as register renaming or instruction number assignment). Unlike the model used throughout this thesis, a block's instructions can be distributed amongst the cores in the composition, with one of the cores being the block owner. As each core can be responsible for a single block, the total number of blocks in flight is equal to the number of cores in the composition. In their work [Kim 07] Kim *et al.* show that a 32 core composition outperforms a single TFlex core by 3x on a set of EEMBC benchmarks and SPEC2000 micro-benchmarks.

The E2 processor is another EDGE based processor that can dynamically compose its cores [Putn 11]. Unlike TFlex, a block is only executed on the core that fetches it, its instructions are not distributed. E2 introduces the concept of segmenting the instruction window into lanes, allowing cores to fetch multiple blocks. This allows E2 to be more adaptable to different block sizes; if the compiler can only generate small blocks, a core can have multiple blocks in its instruction window executing in parallel. For example, if an E2 core has four lanes, each able to fetch a block of up to 32 instructions, it will be able to execute four times as many blocks in parallel compared to a TFlex core when the program being executed is made of blocks smaller than 32 instructions. This means E2 can potentially extract more instruction level parallelism (ILP) from each core in the composition than TFlex can at any point.

Watanabe *et al.* propose a different type of dynamic multi-core processor, where cores share execution units [Wata 10]. The Wisconsin Decoupled Grid Execution Tiles (WidGET) architecture's design is based on a sea of resources, where a core is composed of a simple Instruction Engine (IE) that can send instructions to a set of

available in order Execution Units (EU). This allows for fine-grained reconfiguration of the cores: depending on the available ILP, an IE can increase the number of EUs it needs if there's a high amount of ILP available, or inversely reduce it. Watanabe *et al.* compare the performance of their processor to that of an Intel Atom and Intel Xeon using the SPEC2006 benchmark suite. They show that using in-order cores with a fine-grained reconfiguration can reduce power consumption by 21% compared to the Xeon whilst achieving the same performance. It is even able to outperform it by 26% whilst still reducing power consumption by 6%.

3.1.2 Reconfigurable micro-architectures

MorphCore [Khub 12] focuses on reconfiguring a core for thread level parallelism. It switches between out-of-order (OoO) when running single threaded applications and an in-order core optimised for simultaneous multi threading (SMT) workloads. This provides an opposite solution to our DMP: providing a large core made for ILP that can be modified to better fit TLP workloads. MorphCore outperform a 2-Way SMT OoO core by 10% whilst being 22% more efficient.

ElasticCore [Tava 15] proposes a morphable core that uses dynamic voltage and frequency scaling (DVFS) and micro-architectural modifications such as instruction bandwidth and capacity to adapt the processor to current needs. Unlike heterogeneous systems that can present different sized cores on a single package, the ElasticCore is a single core with four different size configurations. Each size configuration uses more resources than the previous, such as increasing the fetch width from 2 to 4 to 8 instructions. Having all the resources on the single core allows adaptation to be quicker than migrating a thread to a more appropriate core, 1000 cycles compared to the 10,000 for Arm's big.LITTLE architecture.

This is similar to the work of Dubach *et al.* [Duba 13] where micro-architectural features can be modified for better performance or energy efficiency. They provide extensive analysis of SPEC 2000 benchmarks and demonstrate that machine learning and dynamic adaptation can double the energy/performance efficiency compared to a static configuration.

Summary Whilst CoreFusion and Bahurupi use traditional ISAs, they must either support a limited version of composition where a single eight word block is distributed amongst cores (CoreFusion), or add extra instructions which increase code block size by up to 24% (Bahurupi). In the EDGE architecture, instructions are naturally formed

into blocks, and instructions in a block do not communicate via registers; this simplifies the fetching of multiple blocks. Thus TFlex does not need a centralised structure for fetching instructions [Kim 07], however, it does not support the ability of fetching multiple blocks on a single core. Instead, each core can fetch a single block and dispatch instructions on all other cores in the composition. If blocks are small, then the instruction window of each core will never be filled, which in turn reduces the potential performance of core composition. E2 addresses this problem by segmenting the instruction window into separate lanes and allowing each core to fetch multiple blocks. This makes it more flexible than TFlex as each core will have more instructions to execute on average, allowing each core to extract more ILP. Thus, the processor used in this thesis is based on the E2.

3.2 Automated processor reconfiguration

In Ipek *et al.* 's. original work on CoreFusion [Ipek 07] they introduce the *FUSE*/*SPLIT* instructions that allow for dynamic reconfiguration. However, their eight core system only has two possible configurations: either each core executes on their own or they are fused into groups of four cores; no details as to when to compose cores is discussed. This is the same for the original proposal for TFlex [Kim 07], where 5 different configurations are explored (2, 4, 8, 16 and 32 composed cores), but does not provide any insight as to how to determine the composition size automatically.

In the work of Pricopi *et al.* [Pric 14], they show how dynamic reconfiguration is beneficial when it comes to scheduling multiple tasks. However, they do not discuss any method of automatically deciding the optimal configuration beyond a 4 core composition. In their work they use speedup functions determined from profile executions of applications to determine how to schedule tasks. This means that whenever an application is modified, it must be re-analysed to benefit from dynamic composition. They do not discuss what software characteristics help determine when to reconfigure the cores, or how to optimise software.

Gulati *et al.* propose an offline and online scheduling algorithms for TFlex [Gula 08]. This work focuses on maximising speedup for a set of workloads in a multi-tasking environment. The offline model first profiles a program on different compositions and then uses that information to make a decision at runtime whilst their online model simply changes the size of the current composition by +/-1 core and then evaluates whether or not that modification improved performance. In their results they show that the of-

fline profiling tool outperforms the online algorithm. As the threshold is set ahead of time, the online model is not able to fully utilise the system when applications do not meet the required threshold. They demonstrate that a DMP that can adapt to workloads can result in faster response times than a tradition CMP, between 21% to 13x faster.

Summary The main contributions in automatic reconfiguration rely on profiling information to make a decision. This means that new applications will need to be executed multiple times in order to generate a profiling information used by the DMP to decide when and how to reconfigure itself. This procedure can be costly if the number of ways the processor can be reconfigured is high (for example when a programmer must choose between multi-threading and core composition). Whilst Gulati *et al.* propose a runtime solution [Gula 08], this does not *directly* infer a correct composition, instead it changes the composition size by +/- 1 at each reconfiguration interval. They admit that this system is less efficient than profiling.

These contributions lack the ability of determining a how to configure the DMP without the use of profiling information. If a model that can predict how the DMP should be configured to maximise either speedup or energy efficiency without requiring multiple executions of the program, then this would make using DMPs easier. This is why this thesis examines how machine learning can be used to reconfigure the DMP. Machine learning can be used to generate models that *learn* from how programs are affected by core composition to determine the correct configuration of the DMP for any new program. Through this method, the procedure of finding a good configuration can be reduced to a single analysis of the application at hand.

3.3 Code optimisation for EDGE

Whilst there exists no literature on code optimisations geared towards improving the performance of core compositions, some previous work exists on studying how block sizes affect the performance of the EDGE architecture. Smith *et al.* highlight the importance of block size in their work [Smit 06a], stating that larger blocks will lead to better performance. They suggest the use of instruction predication [Smit 06b] that allows EDGE blocks to be fused into a single block, called a hyperblock.

Since EDGE instructions pass their results directly to an instruction's input operands, some optimisations are required to ensure that predicates are efficiently broadcast to all depending instructions. The optimisations are predicate fanout reduction, path-

sensitive predicate removal and instruction merging [Smit 06b]. Overall, hyperblocks are able to improve the performance of a set of EEMBC benchmarks by 29% compared to only using basic blocks. Using the optimisations previously defined improves the performance of hyperblocks by up to 12% compared to non-optimised hyperblocks.

3.4 Improving instruction fetching for composition

Fetching instructions on core compositions can be a challenge, as will be discussed in Chapter 7. This section covers how this issue has been address in previous research.

Robatmili *et al.* [Roba 11] discuss how to improve block fetching for the TFlex processor by selectively refreshing blocks. A block refresh involves keeping the block in the instruction window, flushing the buffer operands and then re-executing the block, skipping the re-fetch. First, they modify TFlex so that blocks are executed locally on the core instead of dispersing the instructions of a block on multiple cores. The two main assumptions made are that a core can only ever execute a single block at a time, and once the block is committed, it can remain in the instruction window. When a new block request is made, the coordinator core in charge of the new block checks to see if any inactive core has that block stored in their window. If it does, then that core is activated and starts re-executing the buffered block. If no idle core has the block in their window, then the coordinator core will ask one of those cores to fetch the block and start executing it. They show that cores may have to store up to 8 blocks in the instruction window to ensure that 75% of the total executed blocks come from a refresh. This technique can improve the performance of a 16 core composition on a set of SPEC 2000 benchmarks by 1.08x on average, with a maximum speedup of 1.16x.

This implementation depends on the concept that cores in the composition currently hold inactive blocks in their instruction window. Whilst this may be the case for TFlex, as it can only execute a single block per core, an E2 type processor is designed to not have idle blocks in its composition. Therefore, another solution to improve fetching for the type of processor used in this thesis is proposed in Chapter 7.

3.5 Hardware techniques for power and energy efficiency

Chapter 6 focuses on changing the size of a core composition at runtime to the energy consumption of single threaded applications whilst still maintaining the same execution times as the fastest static core compositions. This section covers the different techniques for reducing energy consumption using hardware techniques.

3.5.1 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency scaling (DVFS) is a method of modifying the power and energy consumption [Paga 17] by modifying the voltage or the clock rate of the processor. Often times, DVFS is used to reduce energy or power consumption in phases of low performance.

Herbert *et al.* in [Herb 07] demonstrate that DVFS is an effective technique for reducing $energy/throughput^2$ (E/T^2). They suggest two methods of using DVFS: controlling it at a per-core basis, or at a cluster basis, also known as Voltage Frequency Islands (VFI). Using VFIs reduces the design complexity of both the hardware and algorithms that control DVFS. They demonstrate that on a 16 chip multi-core processor (CMP), DVFS can reduce E/T^2 by 38.2% on a set of multi-threaded workloads. The results also highlight how core-level DVFS does not reduce E/T^2 significantly compared to using VFIs: 38.2% for core-level, compared to 37.9% for a 4 core VFI.

Pagani *et al.* consider the use of VFIs in a heterogeneous system in [Paga 17], where cores in a VFI are homogeneous, but the different VFIs are heterogeneous. Their objective is to ensure that tasks are running on the most energy efficient core whilst satisfying the time constraints of the task. Once a task is partitioned and dispatched onto a core in a VFI cluster, Pagani *et al.* use Single Frequency Approximation (SFA) [Paga 13] as the DVFS strategy. SFA determines a single voltage/frequency that satisfies timing constrains of a task. This technique results in an average reduction of energy consumption of 25% and still ensures all tasks finish on time.

Vega *et al.* [Vega 13] underline that whilst DVFS is an effective way of reducing power and energy consumption, the fact that it is decoupled from other techniques such as per core power-gating (PCPG) reduces the overall benefits. They suggest an online algorithm at the OS level that collects data from performance counters and makes multiple decisions based on the data gathered.

3.5.2 Thread migration in HCMPs

Arm big.LITTLE [ARM 13] is an example of a heterogeneous chip multi-core processor (HCMP) that provides two different types of cores to allow the programmer to choose between energy efficiency and performance. A program can be migrated from one core to another depending on the requirements, however this comes at the cost of a very high migration overhead, over 10,000 cycles [ARM 13]. However Gutpa *et al.* show that by selecting the correct core configuration at runtime this can lead to an energy reduction of up to 46%

Haque *et al.* show that an HCMP is able to reduce energy consumption and improve throughput for applications with high-percentile latencies [Haqu 17]. They highlight that service providers receive requests of different lengths of computation. By successfully scheduling the different lengths to the correct cores in an HCMP, they can reduce the energy consumption of short requests by a factor of 50% compared to DVFS.

Adileh *et al.* argue that current proposals for scheduling tasks on small or large cores are inadequate when operating on a power-constrained HCMPs in a multi-tasking environment [Adil 16]. Using linear programming they determine that applications can be ranked based on their delta performance/delta power (DPDP) which is defined as the ratio of performance and power difference between executing an application on a small or large core. Applications that ranked highly are executed on the large core whilst others are executed on the smaller cores. After defining five schemes that used DPDP to schedule tasks within a power budget of 1W per second per application they show that it outperforms other schemes by 16% on average and up to 40%.

Gupta *et al.* tackle the issue of the high number of configurations possible in an HCMP by characterising workloads offline [Gupt 17]. Applications are executed and their performance is recorded on the different available cores, with different parameters tuned such as the core's clock frequency. Then, for different optimisation goals they found the Pareto-optimal configuration for each of the benchmarks. This information is then used to build a classifier that can determine the correct core configuration for a given snippet of an application. Using this classifier Gupta *et al.* are able to improve performance per watt of a set of single and multi-threaded benchmarks by 93%, 81% and 6% compared to an interactive, on demand and powersave governor respectively.

Summary These different techniques show that either dynamically changing some parameters of the hardware (DVFS) or moving where a program is executing (thread migration) is an effective way of reducing energy consumption. However, all these techniques rely on fixed configurations of the processor, such as an HCMP where the core configurations are determined at design time.

3.6 Speculative Execution

Core composition is able to improve the performance of applications by executing many instructions speculatively from the same thread. This section covers work on speculative parallelism, where performance improvements are obtained by speculatively executing multiple tasks in parallel, rather than through deep branch prediction.

Software level The idea of speculatively extracting parallelism at runtime was first introduced by Rauchwerger *et al.* [Rauc 95]. They underline that compile-time analysis of single-threaded programs does not allow for the complete detection of parallel sections, and must be complemented by runtime analysis. They propose a framework for parallelising loops at runtime: instead of detecting if the loop is parallelisable or not, the loop is speculatively executed in parallel and then runtime analysis is conducted to verify that no data-dependencies are violated. If any violations occur, the loop is re-executed serially. Loops must be marked as speculatively parallel by the compiler, the run-time system only verifies whether or not the speculative execution violates dependencies.

Hertzberg *et al.* push the idea of speculative multi-threading further in [Hert 11] by using dynamic binary translation (DBT) to generate optimised parallel code on the fly, they name this the Runtime Automatic Speculative Parallelism technique (RASP). Instead of generating speculative parallel loops at compile time, they propose that idle cores should be used to analyse running programs and generate parallel versions of the loops. The code continues to be analysed even after the generation of parallel loops in order to ensure that the optimal code has been generated. Using RASP, Hertzberg *et al.* demonstrate that their system can lead to a performance increase average of 1.46x on SPEC2006 integer benchmarks and up to a 3x speedup on SPEC2006 floating point benchmarks compared to single-threaded execution.

Whilst the work proposed by Hertzberg *et al.* motivates pairing DBT with speculative parallelism, Koch *et al.* show in [Koch 13] that the majority of the speedup obtained arises from the DBT optimisations and not speculative parallelism. As serial programs can also benefit from DBT optimisations, Koch *et al.* argue that the work in Hertzberg *et al.* does not necessarily motivate dynamic binary parallelisation (DBP) but instead DBT. Without DBT, RASP only provides a 1.12x speedup on SPEC2006 integer benchmarks, compared to the 1.46x when DBT optimisations are turned on. Koch *et al.* underline that the cost of detecting loops which can be parallelised, paired with the cost of starting threads can often outweigh the performance benefits of DBP.

Hardware level Jeffrey *et al.* present a novel tiled architecture called Swarm [Jeff 16] that performs aggressive thread-level speculation. The Swarm architecture executes tasks that are identified via timestamps. Each task can access any data and has the ability to generate new tasks, also known as children, that will be assigned a greater timestamp. All tasks are maintained by a task-queue, and can be executed out of order.

Swarm is extended by Abeydeera *et al.* in their speculation aware multi-threading policy SAM [Abey 17]. They claim that having a high number of speculative threads often leads to a large number of aborted tasks which impacts performance. SAM ensures that tasks with lower timestamps are prioritised as they are most likely to commit, thus reducing the number of aborted tasks. This technique of determining which tasks to run is called issue stage prioritisation. They also relax conflict resolution by using a similar technique to Wait-n-GoTM [Jafr 13]. Instead of assigning tie-breakers to tasks when they are spawned, SAM assigns them when a task acquires a dependence from another task with equal timestamp; this reduces the number of needless aborts. Overall, an 8 in-order core Swarm processor with SAM improves performance by 2.33x compared to a single core for a set of graph algorithms.

Subramian *et al.* present a new execution model that supports nested parallelism and is implemented for a Swarm [Subr 17] . Fractal introduces the concept of grouping tasks into hierarchies of nested *domains*. Tasks in a domain can either execute in order or out of order relative to their timestamps and appear to execute as a single atomic unit to other domains. By allowing tasks to generate domains, Fractal is able to exploit nested parallelism without complicating the software, and can lead to a performance increase of up to 88x compared to Swarm on some graph algorithms.

Summary This section has shown how an alternative form of speculation can be used to improve the performance of applications. This is supported both at the hardware and software level. As seen throughout the section, speculative parallelism requires both hardware and software support to function (SWARM). This is more involved than supporting core composition on an EDGE processor, where the ISA naturally lends itself to deep single-threaded execution speculation.

3.7 Tackling data-dependencies

This section covers the different techniques used to tackle data-dependencies that affect a processor's ability to extract instruction level parallelism, which is the main way core composition improves performance.

3.7.1 Value Prediction

Value prediction is a technique used to speculatively resolve data dependencies between instructions and is used in Chapter 7. This section covers the different proposed techniques, including the one used in that chapter.

The earliest work on value prediction can be retraced to Mikko *et al.* where they propose a *Load Value Predictor* [Lipa 96] that predicts the value of load instructions. They show that performance can be improved by up to 27%.

Perais *et al.* propose the VTAGE *context* value predictor [Pera 14], that adopts the same prediction scheme as the ITTAGE branch predictor [Sezn 11]. Using global branch history, which is easier to maintain that data-flow history [Pera 14], VTAGE can improve performance of some SPEC 2000 and 2006 applications by up to 65%.

The block based D-VTAGE predictor [Pera 15] can quickly issue multiple predictions grouping up predictions as a single block. The basic prediction fetch and update mechanism are inherited from VTAGE, except D-VTAGE is a *computational* value predictor. In order to improve value prediction for tightly knit loops where multiple iterations of the loop body can be live in parallel, the predictor employs a speculative window that is able to keep track of live speculative data. D-VTAGE is able to obtain up to a 1.7x speedup, and averages a 1.10x speedup on a set of SPEC 2000 and 2006 applications.

Miguel *et al.* propose a different technique for value prediction called load value approximation [Migu 14] for applications where value inexactness is acceptable. Applications such as image tracking or image comparison do not need to operate on exact values as they often allow for a margin of error. Therefore, these applications do not need to roll-back on a mispredicted value, and can continue to operate with incorrect data as long as it is sufficiently accurate (an error rate below 10%).

Sheikh *et al.* present a value predictor that is able to avoid mispredictions caused by Load $\rightarrow$ Store $\rightarrow$ Load conflicts [Shei 17]. This is achieved by predicting load memory address at the instruction fetch stage and checking the data cache for that memory address. If the memory address is contained in the data cache, then the value is fetched (this is considered a prediction). If the address is not contained in the cache, then a data prefetch is issued. Their new approach to value prediction generates a 4.8% speedup on a set of SPEC 2000 and 2006, EEMBC and Octane applications. This is almost 2x more than the VTAGE predictor used in the paper (2.1%).

Summary Most of the predictors described in this section focus on value prediction for load instructions. Whilst loads can be slow to execute causing longer data-dependency stalls, the EDGE architecture's reliance on registers for intra-block communication is also a prime suspect. The D-VTAGE predictor is the most adequate solution for this thesis, as will be explained in greater detail in Chapter 7 Section 7.4.2.

3.7.2 Data Prefetching

Whilst value prediction is used to mask data dependencies or long latencies caused by cache misses by predicting values, data prefetching attempts to make data available in L1 caches before it is needed. This can be achieved both in software by inserting instructions that trigger a data fetch-request ahead of time, or in hardware by analysing memory access patterns.

Ainsworth *et al.* [Ains 16] suggest that a stride-based prefetcher is not adequate for irregular memory accesses. They propose a hardware prefetcher that has knowledge on the data structures being treated, to be able to trigger multiple prefetches based on loads snooped on the L1 cache. Using Breadth-First-Search (BFS) as an example with the compressed sparse-row data format, they demonstrate that a specialised prefetcher can improve performance by a factor of 2.3x compared to stride prefetching and software prefetching that only generates a 1.1x speedup.

Prefetching for indirect memory accesses is considered difficult when using a hardware prefetcher [Lee 12, Ains 17]. Thus Ainsworth *et al.* also propose a compiler pass that inserts non-blocking loads to enable prefetching for indirect memory accesses in [Ains 17]. The compiler pass targets loads inside loops whose addresses depend on a loop induction variable and meet a set of conditions to ensure that the prefetches do not cause faults. Once the loads have been detected, the prefetches are scheduled based on a formula they devise in the paper. The formula takes into account the number of loads found in the prefetch sequence for the loop, the position of the real load, and a constant that is based on architectural features such as memory latencies and the possible IPC. Using their compiler pass they are able to improve the average performance of memory intensive benchmarks by 1.1x to 2.7x on different processors.

3.7.3 Register Bypassing and Criticality Detection

Whilst value prediction is used in Chapter 7, there exists previous work on trying to reduce the latencies caused by data-dependencies in core composition; this section describes the solution.

Robatmili *et al.* [Roba 11] discuss the potential bottlenecks caused by data dependencies when executing blocks on large core compositions. This work uses the TFlex processor and presents the Distributed Block Criticality Analyzer (DBCA) that can gather criticality information to optimise the execution of instructions at runtime. The DBCA is able to detect and predict which instructions are late communication edges, that is the last register writes in a block that younger blocks depend on. When a block

is fetched, the core calls the DBCA to get the predicted registers to determine which register writes are late communication edges. Once the values of these instructions are produced they are forwarded to the successive speculative block directly, bypassing the register file. Using this techniques, Robatmili *et al.* show that for a 16 core composition (TFlex processor), the performance of a set of integer SPEC 2000 benchmarks can be improved by 1.08x on average and up to 1.16x at best.

The reason this technique cannot improve performance much more is due to the fact that it only attempts to detect which instructions should use register bypassing. This model can still involve significant latency, as the late communication edges may require data from previous instructions as well, increasing the critical path chain.

Summary Data prefetching and register bypassing can help alleviate the effect of data-dependencies by ensuring that data-dependent values arrive quicker to their destination. Unlike value predictors, data prefetching is a technique that is actually implemented in both compilers and real commercial processors [Inte 16]. Yet both data prefetching and register bypassing do not allow for instructions to execute with speculative data, which, as will be seen in Chapter 7 is better for increasing ILP in large core compositions.

3.8 Dataflow Programming Languages

Chapter 5 explores how a DMP can be used to improve the performance of applications written in a dataflow programming language. This section describes a set of data flow programming languages and what hardware they target.

StreamIt [Thie 02] is one of the first programming languages directed towards streaming applications. As previously described in Chapter 2 Section 2.6, StreamIt defines a set of constructs to build scalable parallel streaming applications. StreamIt was originally intended for the RAW [Wain 97] tile-based architecture, but can be used in other settings as well.

Brook [Buck 04] is another streaming programming language geared towards Graphical Processing Units (GPUs). Unlike StreamIt, Brook extends the C language by providing a new data type and function types. The new data type, called a *stream*, provides a collection of data which can be operated on in parallel, which can be modified by stream functions. A stream function takes one or more stream inputs and will output one or more streams; this is similar to a StreamIt *filter*. To operate on streams in parallel, Brook defines a subset of stream functions called *kernel* functions. These

functions do not have access to global values, cannot call functions that aren't kernel functions as well, and are only allowed to access streams in a read-only *OR* write-only way. This is to facilitate the compiler generation of dataflow graphs for the program.

WaveScript on the other hand is developed for embedded systems that are low powered [Newt 08]. Unlike StreamIt, WaveScript uses an asynchronous streaming model, where the input and output rates of functions are not known at compile time.

Bosboom *et al.* demonstrate how a streaming language can be directly embedded into a more common host-language in [Bosb 14]. They present StreamJIT which inherits StreamIt's programming structure, however the language is embedded inside Java. This allows the new StreamJIT language to benefit from the front-end compiler optimisations developed for Java; a method they call *commensal* compilation.

Summary Brook and WaveScript are both domain specific languages for architectures not explored in this thesis (low powered embedded systems and GPUs). As StreamIt is used to construct scalable parallel streaming applications it makes it a perfect candidate for exploring how to partition multi-threaded applications on a DMP, as seen in chapter 5. Since the current system used does not have a Java Virtual Machine (JVM) it cannot support the commensal compilation technique described by Bosboom *et al.* .

3.9 Partitioning streaming programs on multi-core chip

Chapter 5 explores how partitioning streaming applications on a DMP improves performance. This involves determining which number of threads leads to the best execution time, and how many cores each thread needs. This section covers work on partitioning streaming applications.

Previous work on scheduling streaming applications focuses on finding mathematical ways of partitioning the graph onto the chip [Carp 09, Kudl 08]. In Carpenter *et al.* 's work [Carp 09] they restrain themselves to partitioning a StreamIt application maintaining correctness. Correctness can be defined as a subgraph where the filters are connected. This restriction reduces the number of potential partitions that can be generated by their algorithm and will put TLP in favour of ILP.

Kudlur *et al.* [Kudl 08] choose to represent the partitioning problem as an integer linear programming problem. They start by fissioning stateless filters to obtain the optimal load balance across all cores and assign the filters to a core using a modulo scheduler. Farhad *et al.* also use integer linear programming in [Farh 12] to schedule

StreamIt programs on multi-core. They profile the communication costs of the streaming programs by running the program using different multi-core allocations and feed that information into their integer linear programming model.

Using a machine learning model to partition StreamIt programs was previously explored in the work of Wang *et al.* [Wang 08]. They use a k nearest neighbour (kNN) model to determine the perfect partitioning of a StreamIt program for a multi-core system. Their model is used to find ways of fusing and fissioning filters to discover a new graph that can then be mapped onto a multi-core system.

Summary Whilst this previous work focuses on determining a partition of streaming applications through mathematical models or integer linear programming, Wang *et al.* [Wang 08] show that machine learning can be used. However, none of these proposals explore DMPs, instead focusing on processors with fixed designs. This means that whilst some techniques such as kNN can be used to automate the partitioning decision, there still lacks any exploration of how the thread/core composition design space affects the performance of streaming applications. This thesis therefore conducts that exploration to generate a new model for partitioning streaming applications on DMPs.

3.10 Machine-learning guided performance optimisations

Using machine learning to increase performance has been a popular area of research as of late. There are two areas in which machine learning is used: compiler driven optimisations and runtime driven optimisations.

Dubach *et al.* use machine learning to determine what the performance and energy consumption will be based on the micro-architectural parameters and compiler optimisations that are used in [Duba 12]. Their work explores the 35 MiBench benchmarks by modifying compiler optimisations and micro-architectural features of an embedded system. Over 1000 compiler flags are explored, and 200 configurations of the processor are used (modifying cache sizes, associativity, branch target buffer sizes and associativity). Their design space exploration shows that to obtain the best performance for each of the applications, both the compiler optimisations and processor must be tuned. They then use this information to build a machine learning model using an Artificial Neural Network which is able to predict the performance, in terms of $Energy \times Delay \times Delay$ of an application given a set of compiler flags and micro-architectural features. Their model is able to predict a performance that is close to the best in the space with an error rate of just 3.2%.

Cummins *et al.* developed a deep learning model that takes source-code and is able to learn how the code correlates to performance [Cumm 17a]. Their deep learning model ingests source-code and through a set of transformations, and through training the model, it is able to also generate optimisation heuristics. They train their deep learning model for two scenarios: GPU thread-coarsening and CPU/GPU task partitioning. Compared to state of the art hand-crafted heuristics, the deep learning model is able to outperform both scenarios by 12% and 14% respectively.

3.11 Summary

This chapter underlined that encoding instruction dependency order at the ISA level makes EDGE a more attractive architecture for core composition [Kim 07]. It then showed that of the two EDGE based architectures discussed, the fact that E2 can support a higher number of blocks that TFlex in a composition makes it more interesting, as a higher block count means more ILP can be exploited. The E2 processor is therefore used as a base in this thesis.

Then the different techniques for automatically configuring a DMP were explained. Overall, they rely on profiling applications in order to make decisions meaning that multiple executions of a program are required to create a profile that is used to configure the processor later on. This can be an expensive procedure if the number of configurations is large. This thesis tackles this issue by presenting models that use machine learning to determine the correct configuration of the DMP without requiring multiple executions of the program; making DMPs more practical to use.

This was followed by explaining how block fetching latencies can be reduced to improve the performance of core composition. This work was conducted on a TFlex processor and depends on idle-cores to selectively re-issue blocks on idle cores. Whilst this is a promising approach for a TFlex processor, E2 avoids having idle cores by allowing them to fetch multiple blocks, and thus the concept of instruction buffering is not applicable.

This was followed by exploring different hardware techniques used to reduce the energy consumption of applications, which is a topic approached in Chapter 6. These techniques (DVFS and thread migration) often rely on the processor cores to be designed ahead of time to create performance models, and they can sacrifice some speed to reduce energy consumption. This thesis will show how a DMP can be dynamically reconfigured to reduce energy consumption whilst obtaining the best execution time of a static composition.

Another method of speculative execution was then described: hardware and software techniques that exploit thread-level speculation. These techniques are described to show the breadth of performance optimisations via speculative execution.

Value prediction, data-prefetching and register bypassing were shown to be methods of reducing the impact of data-dependencies between instructions. Even though data-prefetching is successfully applied in commercial products, it does not allow instructions to execute with speculative data. Also, register bypassing and critical detection does not present the best solution to the problem as it is more focused on forwarding values with low overhead. As value prediction allows instructions to execute with speculative data, this is a more promising approach for increasing ILP in core compositions, as will be explained in Chapter 7.

Then, dataflow programming languages, and the different methods used to partition applications written in these languages on multi-core processors was explained. The StreamIt programming language is explored in Chapter 5 as it was designed for architectures similar to the one used in this thesis. The work of Wang *et al.* [Wang 08] motivated the use of machine learning for streaming applications however it focused on homogeneous multi-core processors. This thesis pushes this work further by providing a model that can determine both thread count and number of cores composed.

Finally, two examples of how machine learning has been used for performance optimisations were described. This is to demonstrate that machine learning is becoming a promising method for the domain of improving the performance of programs and determining architectural parameters.

Chapter 4

Setup

Parameter	Values
Decode & Dispatch Width	8
Issue Width	8
Execution Units	8
Number of Lanes	4
Pipeline Depth	5 Stage
L1D cache size	32kB
L1I cache size	32kB
L1 Hit Latency	2 Cycles
Main Memory Hit Latency	120 Cycles
L2 cache size	2MB

Parameter	Values
L2 Cache hit	5-20 cycles
# of MSHR	8
LSQ Organisation	Out of Order
Execution Engine	Out of Order
Branch Predictor	Tournament Style
Branch Latency	3 Cycles
Reconfiguration Latency	100 cycles
Processor Network	Mesh Network
Communication Latency	1 Cycle Hop

Table 4.1: Hardware characteristics of a single core of the processor.

4.1 Dynamic Multicore Processor Simulator

To evaluate the work a customisable cycle-level simulator for the EDGE architecture is used. The simulator is verified against RTL implementation of an EDGE core as described in Putnam *et al.*'s work [Putn 11] and is within 5% from that implementation [Mico 16]. This validation is done by running workloads on RTL and comparing the traces cycle-by-cycle with the simulator [Mico 17].

To maintain a homogeneous view of the system, the same core configuration was used throughout the thesis. The features of the core can be found in table 4.1, and the processor is composed of 16 cores connected via a mesh network (1 cycle hop, Manhattan distance), with the L2 Cache being the shared last level cache. The number of lanes represents how many blocks a single core can hold in its instruction window at a time, and is the same number used in the original proposal of the E2 core (4 lanes) [Putn 11].

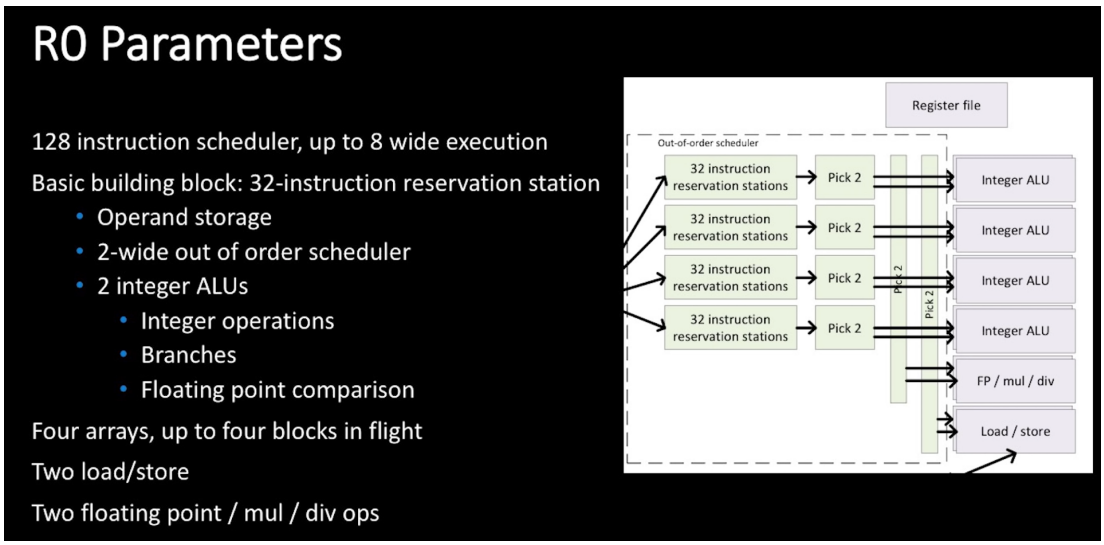


Figure 4.1: Parameters of the R0 core shown by Aaron Smith during the RISE lecture

As of 2018, it is not uncommon to see embedded cores in smartphones that have up to 12 execution units and are able to dispatch up to 6 instructions per cycle [Anan 18b, Anan 18a]. EDGE is designed to reduce the complexity and overhead of out-of-order engines by encoding dependencies at the ISA level [Kim 07, Gray 18a] and thus exploit high amount of instruction level parallelism with less resources. The configuration of the core in table 4.1 represents an example of an EDGE core that could potentially be built today or in the near-future. This setup was in fact described as a ”small” EDGE core by Burger *et. al* during their keynote at ISCA 2018 [Burg 18, Regi 18, Gray 18b].

Chapter 2 Section 2.4.2.2 described how core composition is a lightweight procedure on an EDGE processor. Therefore, for this thesis the basic reconfiguration latency is set at 100 cycles (including the pipeline flush), which is also a latency used in previously published work on core composition [Pric 12]. Since the length of the reconfiguration latency can affect performance, the effect of different latencies is discussed in Chapter 6 Section 6.6. This is the only chapter that uses dynamic reconfiguration.

Previous studies on dynamic multicore processors for EDGE explored chips with up to 32 cores [Kim 07, Gula 08], yet in Kim *et.al*.'s work, they determine that 16 cores composed leads to the best average performance. Both embedded and desktop processors are still seeing their core count rise as time goes on (Apple's A12 features 6, Huawei Mate Smartphone has an 8 core processor, and the AMD Bulldozer features 32 cores), therefore having a 16 core EDGE processor represents a near-future design. Also, having 16 cores available increases the number of possible configurations of the processor which in turn allows for a more impactful exploration of core composition.

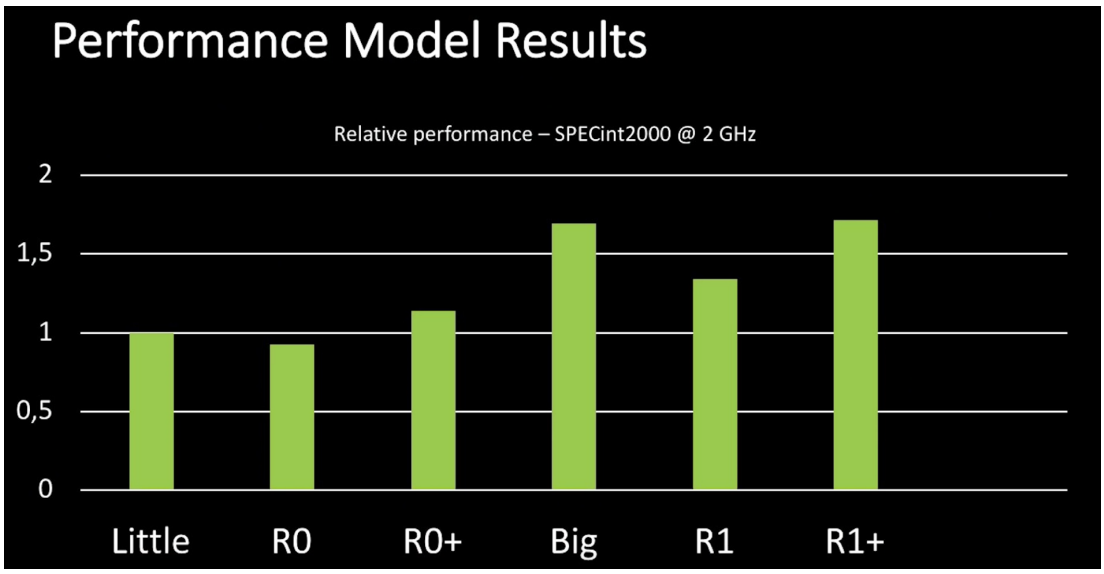


Figure 4.2: Performance comparison of the R0 core with a little (A53) Arm core shown by Aaron Smith during the RISE lecture. The R0+ is a refined RTL model.

4.1.1 Performance baseline

The EDGE architecture aims to improve the efficiency of out-of-order (OoO) processors by involving the compiler more [Euro 19a, Gray 18a]. EDGE is also intended to be used as a general-purpose architecture, able to run operating systems [Euro 19a]. Even though an EDGE based processor lacks traditional OoO hardware such as a register-renaming unit (due to the compiler handling this task), it is still built around a traditional RISC 5-stage pipeline [Gray 18a]. As Arm represents a major player in the RISC space (from micro-controllers to high-performance general-purpose smartphone cores) it presents an important performance baseline for an EDGE core design.

In a presentation at the Research Institute of Sweden (RISE) organised by the Eurolab4HPC project [Euro 19b, Euro 19a] , Aaron Smith presented a performance comparison of an EDGE core using the same parameters as the one used throughout this thesis with different Arm cores. The performance comparison was conducted by Qualcomm using an RTL model of an 8 issue EDGE core (Figure 4.1, and an Arm Cortex-A53 from their SnapDragon SoC [Euro 19a]. To evaluate the performance, the SPECInt2000 benchmark suite was used, and both cores were clocked at 2GHz. As Figure 4.2 shows, a core using the parameters described in Table 4.1 are equivalent to that of a ”little” Arm core clocked at 2GHz.

Audiobeam	Beamformer	Bitonic-Sort	BubbleSort	CFAR
ChannelVocoder	FFT	FFT3	FFT6	FilterBank
FIR	FMRadio	InsertionSort	Matmul-Block	RadixSort

Table 4.2: StreamIt benchmarks used in this thesis.

4.2 Benchmarks

This thesis explores both multi-threaded and single threaded applications to show how dynamic multicore processors (DMP) can adapt to a multitude of situations. This section covers the different benchmark suites used throughout the thesis.

4.2.1 Streaming applications

Chapter 5 demonstrates how configuring a DMP to get the optimal performance out of multi-threaded applications can be learned. To explore this concept, choosing a programming language that naturally exposes parallelism is important as it facilitates the process of generating threads for an application. As the introduction of Chapter 5 will explain, StreamIt is one such language.

The StreamIt repository holds a set of benchmarks that can be used to evaluate the system [T 18]. As the processor and tools used throughout this thesis are still in development, some of the applications would either not compile or execute correctly on the provided simulator; thus a subset of the applications are used. The 15 StreamIt benchmark that worked and are explored in Chapter 5 are shown in Table 4.2. These applications represent a variety of embedded applications and kernels, from digital signal processing to a matrix-multiplication kernel or band pass filters. They can be found in a multitude of devices from digital radios to HDTVs and smartphones (audio/video streaming applications). The benchmarks also present a varying degree of parallelism as will be shown in Chapter 5 Section 5.2. As dynamic multicore processors are intended to adapt to the program at hand, having varying amounts of parallelism in the different benchmarks is essential to demonstrate its flexibility.

4.2.2 San-Diego Vision Benchmark Suite

As core composition is designed to improve the performance of single-threaded applications, it is also important to evaluate a set of serial applications. Chapters 6 and 7 explore a set of Vision Benchmarks designed for hardware and compiler re-

Characteristic	Benchmarks
Memory Intensive	Disparity, Tracking
Computation Intensive	MSER, SVM, SIFT, Localization,Multi NCut
Memory and Computation Intensive	Stitch

Table 4.3: Characteristics of the benchmarks [Venk 09].

search [Venk 09]. The San Diego Vision Benchmark suite (SD-VBS) is composed of nine single-threaded C benchmarks ranging from image analysis to motion tracking. These benchmarks represent state-of-the-art applications in image and vision recognition which are prevalent in embedded systems. The domain of image analysis and vision recognition is prevalent in multiple commercial and research fields, such as robotics, self-driving cars and even facial recognition in smartphones.

Vision applications are usually designed as software pipelines featuring different passes which will naturally form phases throughout the execution of the program. The programs typically have regular and simple control flow which enables the formation of large blocks of instructions. The processor relies on the ability to form large blocks to exploit block level parallelism (BLP) which makes these applications particularly well suited. As the results will show, the phase length has minimal impact on energy savings when the reconfiguration overhead is low.

All the benchmarks in the suite are described here:

- **Disparity** Computes depth information for a given pair of images.
- **Localization** Estimates position of robot based on its surroundings.
- **MSER** Maximally Stable Extremal Regions, a method used for blob detection in images.
- **Multi NCut** Partitions images into conceptual regions.
- **Sift** Scale invariant feature transform is used to extract and describe items found in an image.
- **Stitch** Combines multiple photographs into a single image.
- **SVM** Support Vector Machine.
- **Texture Synthesis** Creates larger image out of a small sample.
- **Tracking** Extracts motion information from a set of images.

and their characteristics in terms of memory/computation intensity are shown in Table 4.3.

4.3 Compiler

All the benchmarks explored in this thesis are compiled using a closed-source EDGE compiler provided by Microsoft. The benchmarks are compiled with *-O2* optimisations as this is the highest level of optimisations available with hyperblock formation turned on (it is its own separate flag).

Chapter 5

Static ahead of time thread and core partitioning

A Dynamic Multicore Processor's (DMP) ability to reconfigure itself allows it to adapt to any program it executes. Whilst being able to reconfigure hardware is a promising approach to optimising execution, DMPs come with their own set of challenges when attempting to finding a good configuration for the program at hand. Given a program that can be parallelised, a DMP can either be configured to run a high number of threads on small groups of cores, a small number of threads on large groups of cores or a heterogeneous mix of both large and small cores. Without deep knowledge of the architecture, knowing what configuration of the processor is correct in order to be able to obtain the best performance, can be a highly time consuming task. This is due to the fact that determining the configuration can require multiple profiling passes if the number of possible configurations is high, and this task will have to be repeated whenever significant modifications to the program are made. This can be further complicated if the programming model does not provide any insights on how the program may be partitioned into threads. The problem of optimising multi-threaded software for DMPs can therefore be split into two distinct tasks. First, finding a programming model that makes software partitioning into threads explicit. Second, using information from both the hardware and software, automate the partitioning of both the software into threads, and the hardware into compositions.

In most parallel programming models, such as OpenMP [Dagu 98], the user is directly responsible for mapping parallelism to the hardware; a difficult and time consuming task [Prab 11]. This is due to the fact that these models extend programming

languages that do not consider parallelism as a defining design factor [Ping 11]. On the other-hand, dataflow programming models such as StreamIt [Thie 02] and Lime [Auer 12] make data and parallelism first class citizens. In these languages, applications are expressed as data oriented graphs and — ideally — the compiler or runtime determines the mapping of parallelism onto the available hardware and controls the grouping of hardware resources. Thus using such a model can be a potential solution to the first part of the problem.

However, optimally mapping parallelism and managing hardware resources remains an open problem given the sheer complexity of the resulting design space. For example, given a 16 core DMP with up to 15 threads, a program can have over 32,000 different configurations of thread to core composition pairings. Rather than exhaustively searching the space, which is a very time consuming task, finding a way to automate the configuration of the processor makes using DMPs more attractive. The number of program features that may influence how to partition programs is large, for example it could depend on the number of tasks, the parallelism made explicit by the language and/or different compiler optimisations. Therefore manually determining a set of heuristics to create a model that selects thread count and core compositions is not recommended as important information may be disregarded. Instead, correlation analysis is used to determine which features, from a set of handpicked features, correlate the most with deciding a good partition and are to be used to generate an appropriate machine learning model.

This chapter analyses how static ahead-of-time reconfiguration of a DMP can improve performance of a set of streaming applications. In this setting, static defines a core composition that does not change during the execution of a program, whilst ahead of time means the configuration is set before execution. These streaming applications include audio signal and image processing and sorting algorithms. Streaming programs are ubiquitous in the embedded systems space [Thie 02] and their mix of parallelism and computation make them an interesting domain for DMPs.

An analysis of the design space is performed and shows the impact of modifying resources and thread mapping and is conducted using a set of StreamIt programs. A machine learning model is developed using the information gathered from the exploration. This model predicts the best number of threads for a given application and an optimal number of cores to allocate to each thread. To demonstrate the viability of the approach the results of the predictive model are compared to the best sampled thread and core composition pairing in a space of more than 32,000 design points. The model

can match the performance of the best sampled points, with speedups of up to 9x on a 16 core processor compared to single-threaded execution on a single core.

The main contributions of this chapter are:

- An analysis of the co-design space of thread partitioning and core composition;
- A study on the impact of a loop transformation on the optimal core composition;
- A machine-learning model that determines the optimal core composition and thread partitioning ahead of time in order to get the optimal performance;
- An analysis of the static code features that are considered the most important for determining a correct configuration of the system by the model.

The chapter is structured as follow, Section 5.1 motivates this work by showing the complexity of the design space. Section 5.2 describes the methodology and section 5.3 presents an in-depth analysis of the design space. Section 5.4 develops a machine-learning model to predict the best thread mapping and core composition whilst section 5.5 shows the performance of the model. Section 5.6 concludes this chapter.

5.1 Motivation

5.1.1 Finding an optimal configuration

This section motivates the difficulty of finding a good combination of thread and core partitioning. First, a simple experiment is conducted where the *FilterBank* StreamIt benchmark is analysed using a 16 core dynamic multi-core processor. *FilterBank* distributes its inputs amongst an array of discrete Fourier transform (DFT) filters, the outputs of the DFT filters are down-sampled, up-sampled and then recombined to form a processed signal [T 18]. The program's tasks are partitioned into threads and a varying number of cores are allocated to each of the threads. Since one of the threads is the master which creates and joins all worker threads, this means that the application can be partitioned in up to 15 threads, and 15 cores can be used for those threads. It also means that the single-threaded version of an application is in fact one master thread and one working thread. To clarify this situation, the term single-working-thread is used throughout this chapter.

Choosing the correct composition topology for each thread is equivalent to the knapsack problem and is therefore NP hard. Thus, to reduce the total design space, this chapter assumes that each composition can only be formed by cores that follow

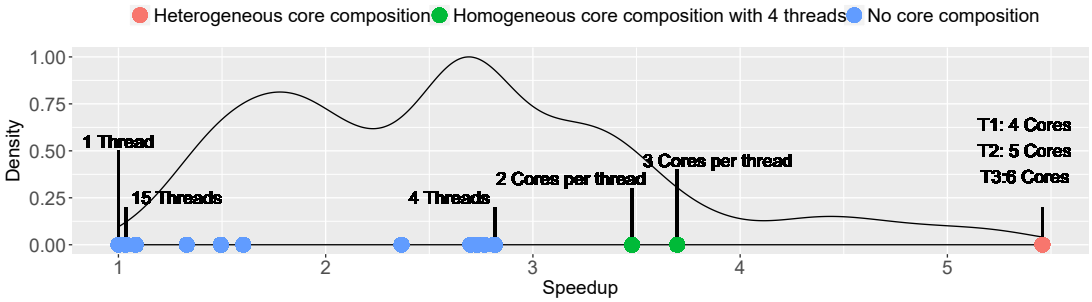


Figure 5.1: Distribution of speedups for FilterBank when using different core composition and thread pairings compared to single-core single-working-thread. The dots on the X-axis represent specific configurations.

each other in row order. Whenever a composition is started, the master core will always be the core with the smallest core ID. These constraints may lead to sub-optimal performance as other core mappings and master cores may be more suitable in certain situations. Determining which topologies are optimal is left as future work.

As each thread must have at least one core assigned to it, and not all cores have to be grouped together, the total number of configurations are:

$$15 + \sum_{threads=2}^{15} \left(\sum_{cores=threads}^{15} \frac{(cores-1)!}{(threads-1)!((cores-1)-(threads-1))!} \right) \tag{5.1}$$

In Equation 5.1, the constant 15 represents the 15 different number of cores that can be composed for the single-working-threaded version. There are 32,767 combinations (exhaustive space) of thread mappings and core compositions pairings that can be generated. In this chapter, a design point represents one of these 32,767 different configurations.

Figure 5.1 presents the speedup density distribution from a subset of the co-design space of *FilterBank* as a density graph where the Y axis represents the density normalised by the total number of design points in the space. The speedup is measured by comparing the performance of the design point to that of a single-core/single-working-thread. A single-core/single-working-thread is used here as it represents the default ”unmodified” configuration. For this experiment, 1316 different thread/core combinations are explored; the reason this number is chosen is explained later on in section 5.2.

As can be seen in figure 5.1, the majority of the design points result in a speedup of 2.7x. The best speedups however are far fewer than the average case (4x smaller density) and can improve performance by up to 5.5x. Thus, if a good configuration is found, this can yield an important speedup compared to the average configuration.

However since the number of good configurations is low, this underlines the notion that finding a combination of threads and cores is a non-trivial endeavour, as randomly choosing a configuration will result in a sub-optimal performance. Indeed, even if the average case is 2.7x faster than a single core, Figure 5.1 shows that there exists a good number of configurations that can lead to less than average speedups.

5.1.2 Minimising the search space

Whilst there exists a large variety of thread-core combinations, certain design choices can be used to try to minimise the space. For example, choosing to only do multi-threading reduces the search space to 15 possible solutions whilst only combinations that lead to homogeneous core compositions reduces the search space to:

$$\sum_{threads=1}^{15} \lfloor \frac{15}{threads} \rfloor = 45 \tag{5.2}$$

where the constant 15 represents the number of cores available. Using only homogeneous core compositions, which facilitates the core partitioning decision, would therefore lead to 45 possible solutions. However, reducing the search space limits the potential obtainable speedup. Figure 5.1 shows the performance distribution of these specific design points. Their location on the X-axis represents the speedup obtained for that specific configuration. The points represent a set of different design choices such as only using multi-threading, using homogeneous core-compositions with threads and using heterogeneous core-compositions with threads.

Using only multi-threading can lead to some performance improvements, however it will not result in the optimal performance. For the *FilterBank* benchmark, 4 threads leads to the fastest execution time when only using multi-threading. This performance is in the highest peak of the density curve, which means that finding the best number of threads for the benchmark will only lead to average performance improvements in this case. However, using too many threads, such as 15 threads, leads to a degraded performance compared to the average. This is due to the fact that the communication overhead between threads will be too high.

To explore homogeneous core composition, the optimal number of threads, which is the number of threads that leads to the fastest execution time without core-composition is used as a baseline. In this case only 2 homogeneous pairings exist: 2 cores fused for each of the 4 threads or 3 cores fused for each of the 4 threads. Figure 5.1 shows that homogeneous core-composition will outperform only using multi-threading by 1.3x,

however it is not the optimal solution. In the end, using a heterogeneous configuration leads to a 1.5x speedup compared to the fastest homogeneous configuration. Therefore, it is important to consider all possible configurations to ensure the possibility of obtaining the best performance.

5.1.3 Summary

This section shows that multi-threading with heterogeneous core-composition is the optimal solution. This means that the total space must be explored in order to ensure that the best speedup can be found. Due to the size of the space and the fact that there can be no a priori about good configurations, using machine learning to predict configurations is a promising approach. By exploring a set of StreamIt benchmarks a machine learning model can learn what features correlate with the correct configuration. This example illustrates the necessity for designing the technique to predict both the optimal number of threads and core composition to use. The next section will present a more in-depth analysis of the design space before presenting the machine-learning predictive model.

5.2 Methodology

This section describes the setup used throughout this chapter to conduct the design space exploration. It starts by presenting the overall work flow and then explores briefly some of the features of the benchmarks. Finally the section explains how the number of design points used throughout the exploration were determined.

5.2.1 Overview

Figure 5.2 presents the work flow of the system used in this chapter and Figure 5.3 illustrates the work flow on a synthetic StreamIt graph. First, the source-to-source StreamIt compiler is used to unroll loops as this is often beneficial when cores are composed as will be seen later in Section 5.3. Then, static code features such as the program's graph structure are extracted from the StreamIt code through the StreamIt source-to-source compiler. These features are used as an input to the first machine-learning model that determines how many threads will be required based on an estimate of Thread Level Parallelism (TLP) found in the program. This information is used to partition the program into threads which is done by the StreamIt compiler

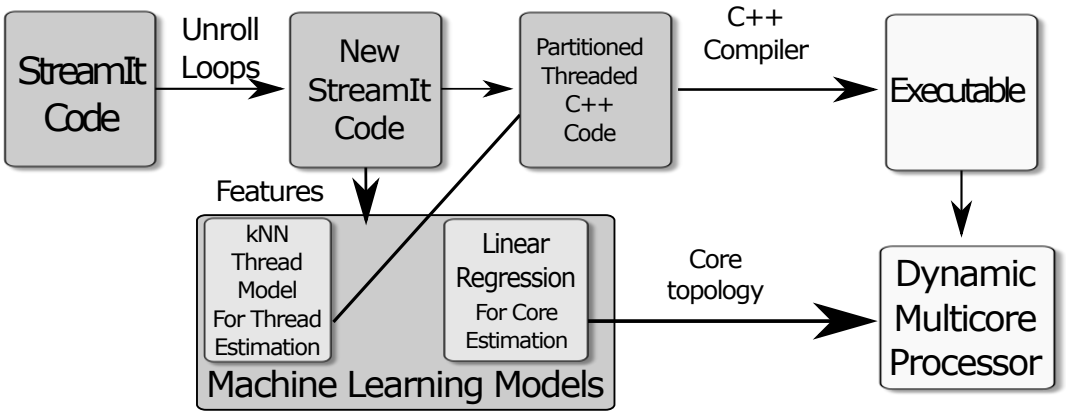


Figure 5.2: Description of the work flow. Two distinct machine-learning models are used to predict the optimal thread partitioning and core composition based on static code features.

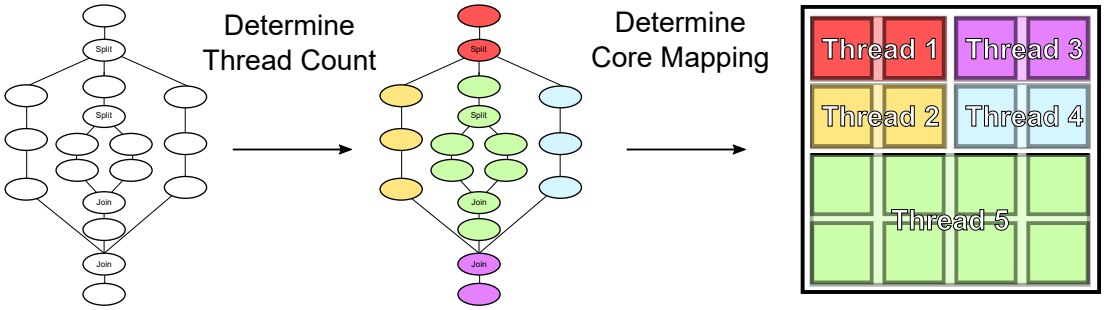


Figure 5.3: Example of a StreamIt program being partitioned into threads (represented by the different colours) followed by assigning cores to each thread.

which produces a C++ program using pthreads. This is exemplified in Figure 5.3, the colours filling in the nodes represent the threads each node has been assign to. This C++ program is then compiled using the compiler for EDGE described in Chapter 4 Section 4.3.

Then, a second machine-learning model is deployed which also analyses static code features extracted from the SteamIt code, once again provided by the source-to-source compiler. This model decides on the number of cores each thread will have. This is achieved by estimating the amount of Instruction Level Parallelism (ILP) that can be possibly extracted in each thread and by determining how many physical cores should be fused for that thread. Finally, the processor is reconfigured to compose the requested resources ahead of time and execute the partitioned program. For example in Figure 5.3, once the graph is coloured, the machine learning model estimates the potential ILP in each group and assigns a number of cores each thread will execute on.

	Audiobeam	Beamformer	Bitonic-Sort	BubbleSort	CFAR
# Of Executed Inst	650K	3.9M	1.3M	230K	770K
	ChannelVocoder	FFT	FFT3	FFT6	FilterBank
# Of Executed Inst	17.5M	1M	588K	595K	858K
	FIR	FMRadio	InsertionSort	Matmul-Block	RadixSort
# Of Executed Inst	947K	926K	100K	1.3M	153K

Table 5.1: Number of dynamic instructions each program takes to execute with default inputs and iteration count set to 10.

5.2.2 Design Space

The benchmarks used throughout this chapter are shown in Chapter 4 Section 4.2.1. These applications represent a variety of embedded applications and kernels, from digital signal processing to a matrix-multiplication kernel or band pass filters. Table 5.2 shows the number of filter instances and SplitJoins for each of the benchmarks. As a refresher from Chapter 2 Section 2.6.1, SplitJoin filters are functions which distribute and collect data from parallel filters. The applications feature a different number of SplitJoins which determine the task-level parallelism. This is to include a variety of situations to test the flexibility of the dynamic multi-core processor. Whilst SplitJoins often facilitate the decision of how to partition the programs into threads, they are not the only way to exploit thread level parallelism. The applications which do not feature SplitJoins can still be split into threads and will operate in a pipelined fashion [Thie 02]. For each benchmark the default inputs provided in the repository [T 18] are used and the default iteration count is set to 10. An iteration count defines the number of times the entire application is executed; in this case, each of the benchmarks is fully executed 10 times. Table 5.1 gives the total number of dynamic instructions executed for each of the applications.

The parameters and size of the space are given in Table 5.3. In this study the 16 core DMP defined in Chapter 4 Section 4.1 is used. Having 16 cores allows for a larger variety of configurations, this also represents a processor similar to a tiled embedded system such as Tilera or Raw. All cores in the DMP are utilised; Core 0 is assigned to the main thread and for runtime management. This leaves 15 cores available for each application. Each core is restricted to running only a single thread, as no context switching is supported, which leads to a possible number of threads between 1 and 15. The core-composition is not used on the master core, leaving 15 physical cores to be

Type	Audiobeam	Beamformer	Bitonic-Sort	BubbleSort	CFAR
Filter Instances	18	56	82	18	3
# of SplitJoins	1	2	44	0	0
Type	ChannelVocoder	FFT	FFT3	FFT6	FilterBank
Filter Instances	53	20	185	99	67
# of SplitJoins	1	12	44	96	9
Type	FIR	FMRadio	InsertionSort	Matmul-Block	RadixSort
Filter Instances	131	29	6	4	13
# of SplitJoins	0	7	0	7	0

Table 5.2: Number of filter instances and SplitJoin filters present in each benchmark.

Parameter	**Values**
# of cores in the processor	16
# threads per application	1 – 15
# cores per thread	1 – 15
# sampled core compositions	100
# our sampled space	1316
# total sample space	32767

Table 5.3: Design space considered per application.

distributed to each of the worker threads. Cores can be composed together to form a composition with up to 15 physical cores, making the total number of cores assigned to a thread between 1 and 15. Of course, not all cores have to be assigned to a thread, in this case all remaining cores that aren't in a composition or a thread are turned off. Overall, this leads to a total space size of 32,767 unique combination per benchmark as previously described in Section 5.1.

5.2.3 Sample Space

Given a partition, any benchmark that can be split into 15 threads requires 32,767 executions to cover the entire space. Running the exhaustive exploration of the space for a benchmark requires approximately a week of simulation on a 572+ node supercomputer. For this reason, a sample of 1,316 random points from the entire space is utilised.

This roughly corresponds to 100 core compositions for each number of threads; the actual number, 1,316 is smaller than 1,500 since for low and high thread counts there are fewer than 100 possible different core compositions. For example, a single-working-thread can have only up to 15 different core-compositions (1 through 15) whilst 15 threads can only have a single core given to each thread. *InsertionSort* is the only exception since it can at most only be split into 5 threads leading to 415 sample points.

To generate the random space, 100 core to threads mappings are generated (when possible) for each number of worker threads (1 to 15). The only restrictions placed on the composition topologies are: the total sum of cores must not surpass the available physical core count (15), each thread must have at least one core and a composition must contain cores that are in sequential row order. The configurations generated do not have to utilise all available cores in the system, unused cores can simply be turned off. Overall, the space exploration requires one week of simulation on a supercomputer [Edin 16].

To gain confidence that the best configuration from the sample space is indeed close to the real best in the entire space, a statistical model based on the Early Stopping Criterion [Vudu 03] is deployed. This statistical model estimates, given a sample of the total space, if the best observed performance of that sample space is within a percentage of the statistical best performance, a more detailed explanation can be found in Chapter 2 Section 2.8. The results demonstrate that the sample space selected is representative of the whole space.

Figure 5.4 shows, for each of the benchmarks, the proximity to the statistical best when increasing the sub-sample space given a maximal uncertainty of 5% (i. e. minimum 95% confidence). As can be seen by the plain line, the model shows that the best sample point is actually within 5% (0.05 proximity) of the best for all the benchmark. The proximity is measured by taking the best currently observed point and comparing it to the latest discovered point. To further prove that the statistical model based on the Stopping Criterion is indeed accurate, an exhaustive exploration of five benchmarks is conducted. The dotted line in figure 5.4 shows the actual proximity to the best for *Audiobeam*, *Beamformer*, *BitonicSort*, *CFAR* and *FMRadio*. As can be seen after 1316 samples, the achieved performance is actually very similar to the one predicted by the statistical model, hence confirming prior work [Vudu 03]. To summarise, it can be concluded that the best point found in the sample space of 1,316 points is at least within 5% of the real best in the exhaustive space with 95% confidence.

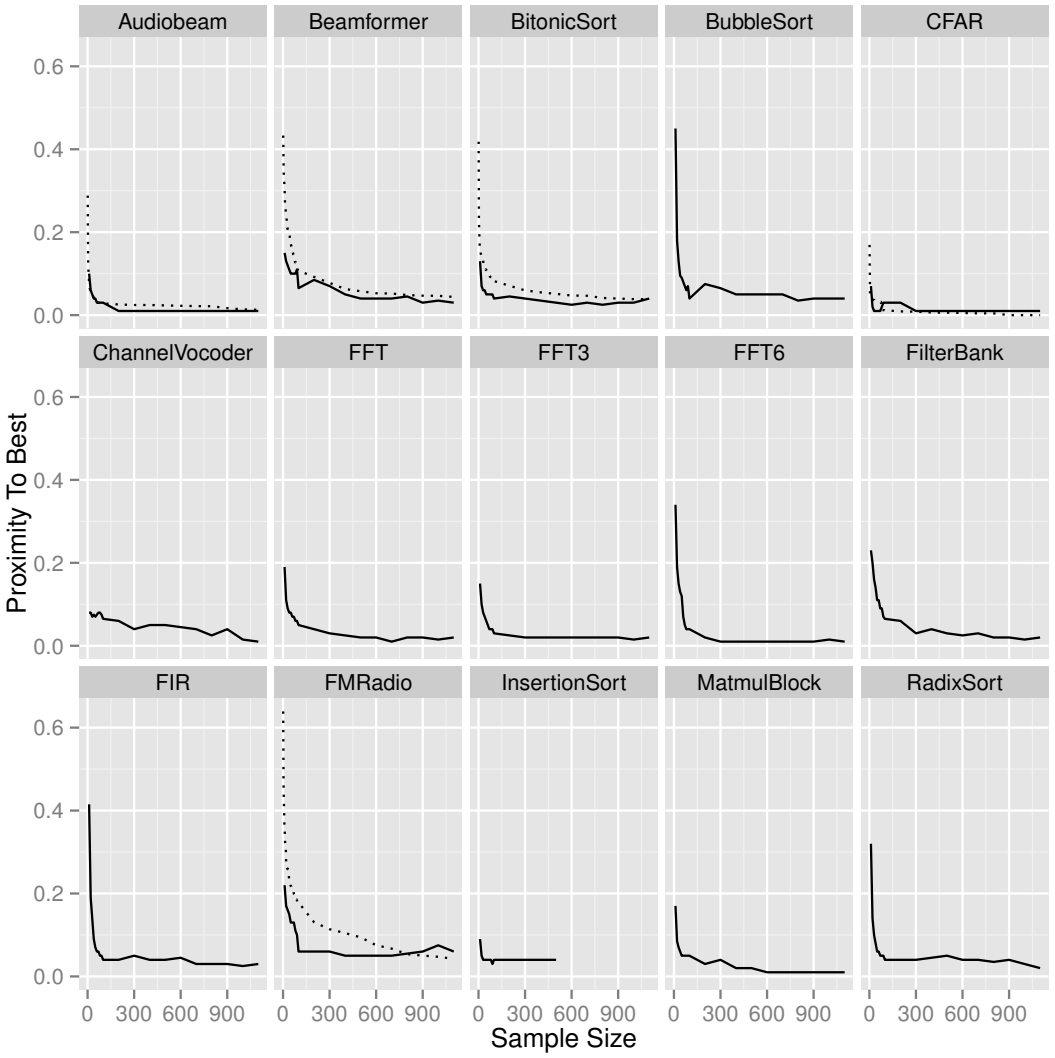


Figure 5.4: Statistical (plain line) and actual proximity (dotted line, this is only done for 5 benchmarks) to best performance using a subset of the sample space.

5.2.4 Synthetic Benchmarks

One of the difficulties of building a machine learning based model for StreamIt is the lack of a large number of benchmarks available [Wang 08]. To overcome this problem, generating synthetic benchmarks can be a solution [Cumm 17b]. Thus synthetic StreamIt benchmarks are generated and statistics are gathered from them in a similar style as in [Wang 08]. In this chapter, 1000 synthetic benchmarks are used to build a model for predicting the number of threads, which will be described later in section 5.4

To ensure that the synthetic benchmarks are representative of realistic benchmarks they are created using filters from a set of example StreamIt programs found in the ex-

	Av. Number of SplitJoins	Average Number of Filter Instances
Selected Benchmarks	14	52
Synthetic	22	64

Table 5.4: Data on the synthetic benchmarks compared to the selected benchmarks

ample directory in the repository. 30 different possible filters with different incoming and outgoing rates and different inputs and outputs types are used to increase the variety of the dataset. To ensure that the synthetic benchmarks are similar to real benchmarks, the total number of filters and split joins are within the average of the realistic benchmarks. Table 5.4 gives the average number of SplitJoins and filter instances for the synthetic benchmarks vs the real benchmarks used in this chapter. As can be seen, the synthetic benchmarks, on average, have more SplitJoins than the real benchmarks; this is due to the fact that a few of the benchmarks presented in the chapter don't have SplitJoins at all which can quickly reduce the average. Since these benchmarks are built to be used for predicting the number of threads an application should use, and SplitJoins are explicit declarations of task-level parallelism, having a higher average number of SplitJoins is not detrimental to building the model.

5.3 Design Space Exploration

This section describes the exploration of the software/hardware co-design space. The software side includes partitioning the program, determining the number of threads and the specific source-level optimisations. The hardware side is about finding out the best core composition that maximises performance for a given partitioning.

5.3.1 Thread Partitioning

In this section, the term optimal number of threads defines the number of threads that results in the best performance for any given benchmark. Thread partitioning is about deciding how many threads to create and how to partition filters into these threads. To simplify this study, the default streaming partitioner is used to decide on how to allocate filters to threads which is based on simulated annealing [Kirk 83].

On the hardware side, two scenarios are considered: the "without composition scenario" where there is exactly one core per thread and the "with composition scenario"

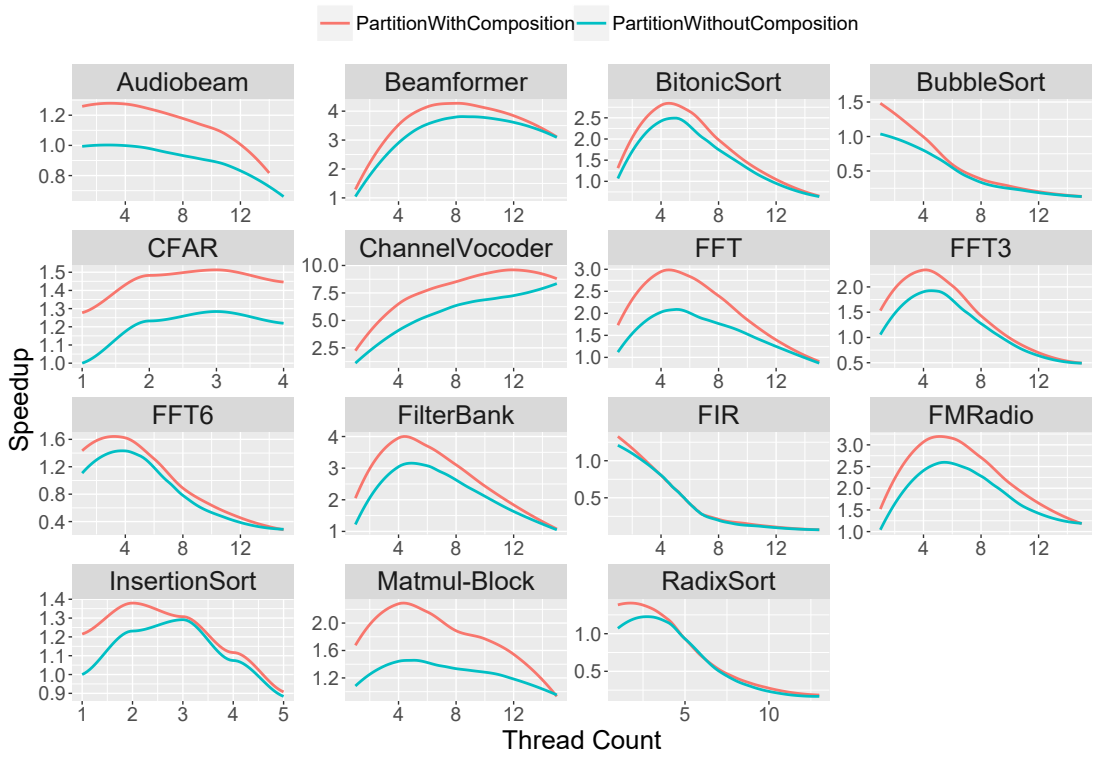


Figure 5.5: Speedup obtained when increasing the number of threads with and without core composition. Baseline is single core single-working-thread. Higher is better

where each thread receives between 1 and 15 cores. Figure 5.5 plots how performance varies under both scenarios as a function of the number of threads. Not all programs can be split into 15 threads, which is why not all curves meet at $x = 15$. In this figure, the "with composition scenario" uses points from the sample space that result in the fastest execution time for a given number of threads. Once again, performance is measured by comparing the execution time of a design point to that of the single core/single-working-thread as it represents the default configuration

Overall the optimal number of threads using core composition is very similar to the scenario without composition as both curves follow the same performance trends. This is due to the fact that StreamIt is oriented towards task-level parallelism and thus, multi-threading is a natural fit for performance improvements whilst core-composition may have less of an effect. As both scenarios follow the same trends the optimal number of threads for a benchmark can be estimated independently from the hardware composition. Also, since the number of cores in a composition is decided based on the amount of instruction level parallelism estimated in a thread, as will be explained in Section 5.4, the thread count must be chosen before deciding on the number of cores that will be allocated to each thread. The system can therefore proceed in two stages:

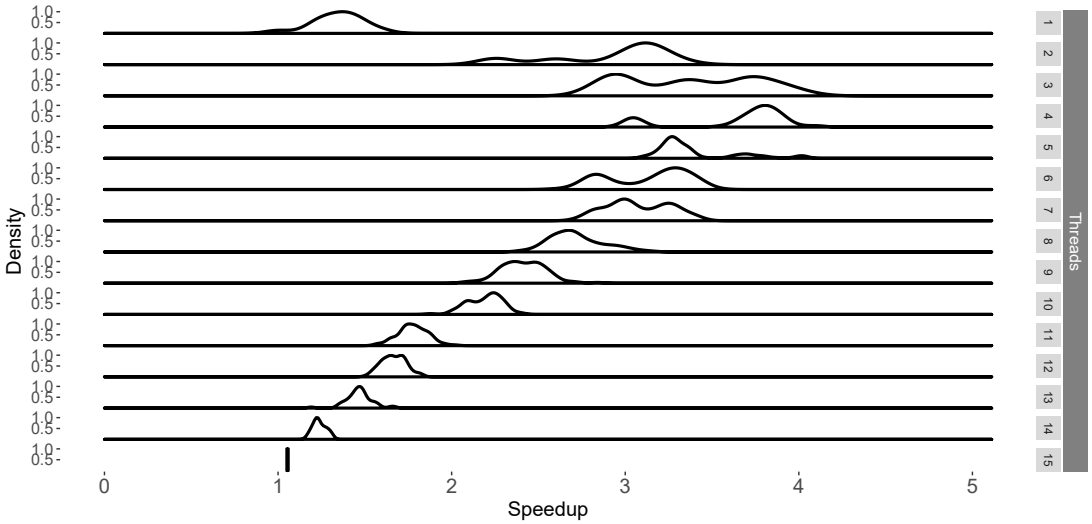


Figure 5.6: Distribution of FilterBank performance when modifying the number of threads and compositions. Speedup is obtained by comparing performance of configuration to single core, single-working-thread.

first determine the optimal number of threads and then decide on a core composition.

Figure 5.5 also shows that the performance of most benchmarks starts to deteriorate past a certain number of threads making it critical to not over-allocate threads. Since the optimal number of threads varies between benchmarks, there is no general thread count that can be applied to all of them. By conducting a feature analysis of the set of applications, a model can be built to determine the optimal thread count for each of the benchmarks. This procedure lends itself well to the use of machine learning. Finally it is important to observe that executions without compositions always perform worse and thus it is essential to consider composing cores to get the optimal performance.

5.3.2 Core Composition

Using core composition, the processor fuses a number of cores and associates them to a thread to increase its performance. Whilst this flexibility is advantageous, choosing the right number of cores for a given thread is difficult due to the large number of possible configurations [Gula 08].

Figure 5.6 shows how multi-threading and core composition affect performance for the *FilterBank* benchmark. The curves represent the density distribution of speedups, compared to single core/single-working-thread, for different core compositions as a function of the number of threads. The right hand side Y-axis represents the number of threads present in the current version of the benchmark whilst the left Y-Axis repre-

sents the density normalised by the total number of points in the design space. For each of the thread counts the benchmark is executed with 100 different core-compositions. The density curve for thread 15 is a single point as there exists only a single composition, so a line is drawn to represent where that point lies. Whilst only *FilterBank* is shown here, a figure for each of the benchmarks was generated and can be found in the Appendix Figures A.1 through A.14

The width of each of the curves represents the influence of composition on *FilterBank*'s performance for a given number of threads. For this benchmark, the impact of having core-composition enabled often leads to a 1.5x speedup compared to running only in multi-threaded mode; this can be seen for 1 to 4 threads. Interestingly, as more threads are used, performance worsens, echoing the results shown in the previous section. This is due to the fact that when the number of threads is increased, synchronisation between threads will increase whilst the potential number of cores which can be fused decreases. In the case where the application does not feature highly parallel tasks, de-prioritising core compositions can negatively impact performance. This signifies that for the benchmark *FilterBank*, it is more important to compose cores with a small number of threads rather than add more threads to the application.

5.3.3 Impact of Loop Transformation

Composing cores exploits instruction parallelism by running multiple EDGE blocks on a composition. As physical cores in a core composition must communicate to submit block address predictions, and commit information to each other, having a small number of blocks will reduce the communication overhead. Since physical cores can fetch more than a single block when the blocks are small, if the program being executed is comprised of mainly small blocks this will cause a composition to fetch very frequently. Thus finding methods to increase the average size of the blocks can lead to reduced overhead which improves the core composition's ability to exploit instruction level parallelism (ILP). One method of increasing the size of the blocks is through loop unrolling, and thus the impact of loop unrolling is studied on the benchmarks.

In this Chapter, unrolling is done at the source level via a flag passed to the StreamIt source-to-source compiler, using an unrolling factor of 32 when possible. Given a number of times the loops must be unrolled, the StreamIt source-to-source compiler will generate the multi-threaded C++ code with the loops unrolled. Figure 5.7 presents an example of how loop unrolling affects performance on the *FilterBank* benchmark.

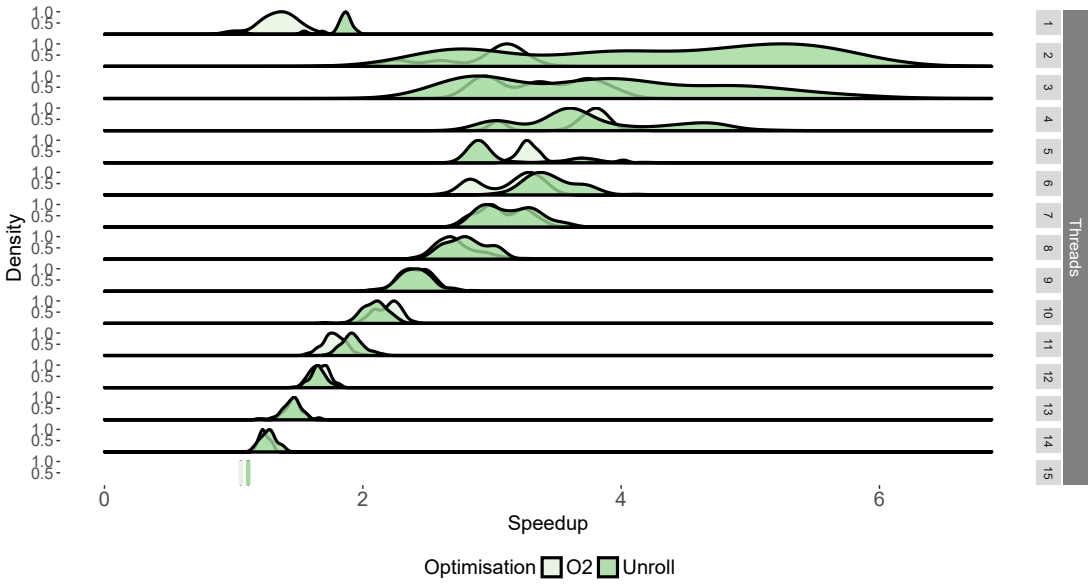


Figure 5.7: Distribution of FilterBank performance when modifying the number of threads, composition and unrolling factor. Speedup is obtained by comparing performance of configuration to single core, single-working-thread without unrolling.

The graph presents the same information as Figure 5.6 but comparing O2 optimisations with O2 and Unrolling. Figure 5.7 shows that unrolling loops for *FilterBank* can improve performance by up to 1.42x compared to the fastest non-unrolled version.

Figure 5.8 shows how unrolling affects the amount of speedup obtained by running each of the StreamIt benchmarks on a single-working-thread using different number of cores in the composition. The X axis represents the number of cores in the composition, ranging from single core to 15 whilst the Y axis compares the execution time in number of cycles for the benchmark using a single core vs. a given core composition. The colours of the lines represent with and without unrolling. As can be seen, five benchmarks benefit from unrolling, these are *Beamformer*, *ChannelVocoder*, *FFT6*, *FilterBank* and *FMRadio*. Figure 5.9 complements Figure 5.8 by showing the speedup obtained by unrolling loops compared to no-unrolling when executing the benchmark on a single core. On average, the information shown in Figure 5.9 and Figure 5.8 coincide: benchmarks that don't scale see no difference in performance when loop unrolling is called.

For the benchmarks that do not scale with unrolling; this is most certainly due to the for loops containing conditional statements which may keep the block size small. When a loop that holds multiple conditional statements is unrolled, conditional statements may not be fused into a single block; thus the block size does not change. This

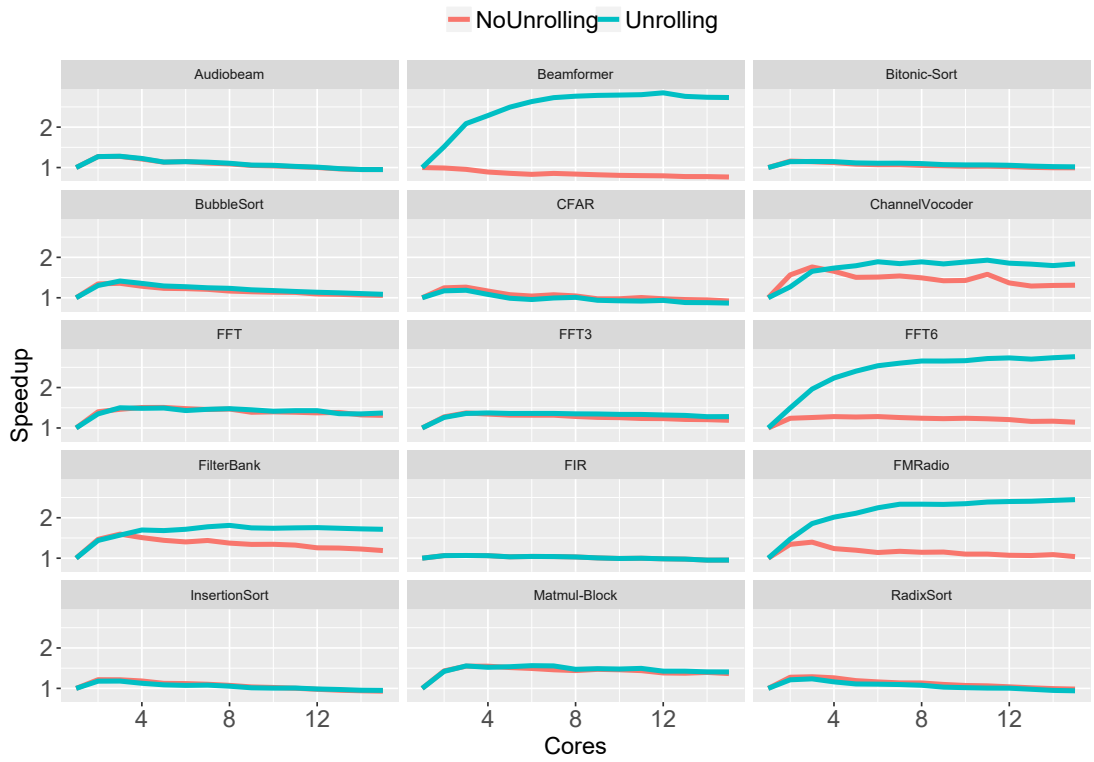


Figure 5.8: Effect of unrolling on performance via core-composition on the single-threaded versions of each benchmarks. The baseline is single core single-working-thread.

is due to the fact that the compiler cannot generate multiple instruction predicates per block. Benchmark *FMRadio* sees a 3x improvement compared to the non-unrolled version, this is due to the fact that all the loops are fully unrolled, reducing the total number of instructions required to execute the task. For the *FFT6* benchmark, unrolling loops will actually cause the single-core version to be slower than its not unrolled version. This is due to the fact that for *FFT6*, the source to source unrolling adds intermediate variables in the loop which increase the number of loads and stores. Whilst it may be slower on a single core, as seen in Figure 5.9, having a core-composition running the thread will still result in faster execution than without loop-unrolling.

Figure 5.10 shows the influence of loop unrolling on the average size of an EDGE block for each of the benchmarks. The size represents the number of instructions executed in each of the blocks. The data for *Beamformer* and *FFT6* in Figure 5.10 corroborate the idea that larger block sizes results in better performance when composing cores. For these benchmarks, unrolling increases the average size of the block by at least 3x, which in turn improves the performance of larger compositions as seen in Fig-

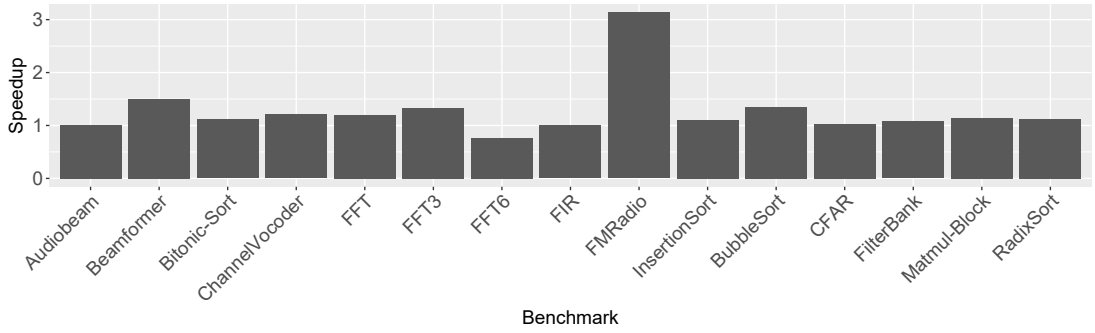


Figure 5.9: Speedup obtained when executing on a single core with loop unrolling compared to without. Higher is better.

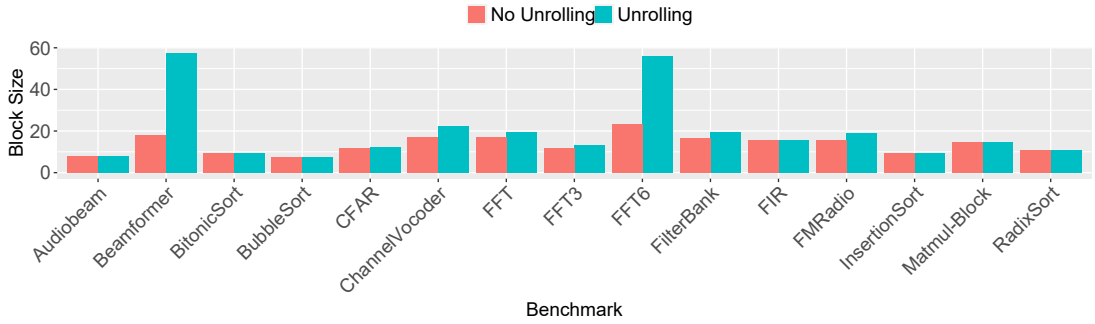


Figure 5.10: Average size (in instructions) of blocks executed with and without unrolling for each benchmark .

ure 5.8. Benchmarks *ChannelVocoder*, *FilterBank* and *FMRadio* also see an increase in block size yet it is not as important and averages out at a 1.22x increase. That said, even a small increase improves the ability of using core composition more efficiently.

Overall, this section has shown that unrolling can improve performance by increasing the size of blocks which helps improve the efficiency of core-compositions.

5.3.4 Co-Design Space Analysis

This section presents the results of the entire co-design space exploration. Figure 5.11 characterises how much of a performance increase, over a baseline of a single-core single-thread with O2 optimisations, can be obtained with and without unrolling. For each benchmark, the *THREAD* bar represents the maximal speedup obtained by dividing the program into threads and assigning one core per thread. The *CORE* bar represents the best speedup when the benchmark is executed on a single-working-thread and fuse all cores. *BOTH* represents the best speedup obtained for each benchmark using a combination of *THREAD* and *CORE*. Finally, for each benchmark, the results are obtained for both an unrolled and not unrolled version to compare how the compiler optimisation affects performance. Figure 5.11 shows that when loops are not unrolled,

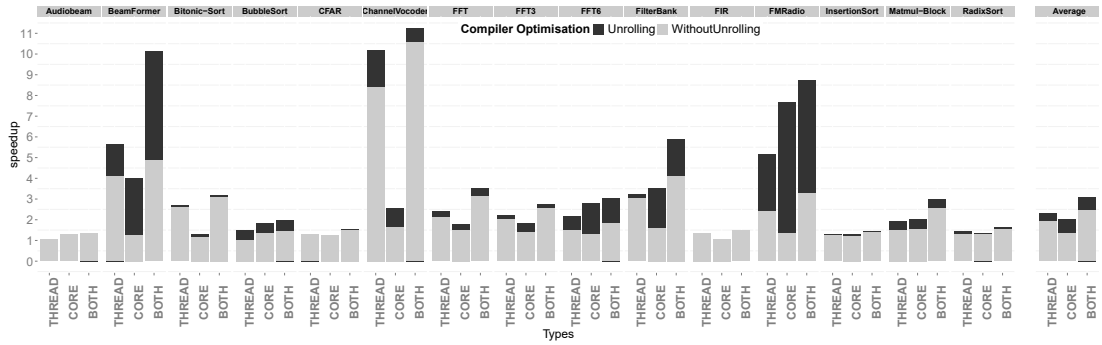


Figure 5.11: Speedup obtained by choosing best core composition, best thread number and the combination of both optimisations. The baseline for the speedup measurement is single core, single-working-thread execution using O2 compiler optimisations. Higher is better.

composing cores will not greatly improve performance. This is due to the fact that the amount of ILP found in filters without the unrolling is too little for there to be any benefit of composing cores.

In the scenario where there are no specific optimisations for composition, multi-threading will be the main source of performance. This can be seen when studying the average (geometric mean),without unrolling. Finding the optimal number of threads gives a speedup of 1.92 compared to 1.33 when using only core composition, which is an improvement of 44%. This changes when taking unrolling into account as the core compositions can be used more efficiently. In this case, the speedup obtained from composing cores without multi-threading is only 13% worse than using only threads. For the *FMRadio* benchmark, unrolling makes using only core-composition better than using multi-threads without core composition. This information corroborates with the data seen in Figure 5.8; it presents a unique case where the effect of core composition is important enough to change the dominant performance enhancer. The performance increase obtained via the source-level loop unrolling via the compiler demonstrates that some modifications to the code must be done to ensure optimal use of the dynamic multi-core processor.

Overall the results show that for these applications, multi-threading leads to more performance gains than using core composition. This is natural as StreamIt applications are naturally geared towards thread level parallelism (TLP) as most programs have at least one SplitJoin as seen in the Table 5.2 which gives the number of split-joins per benchmark. Benchmarks with SplitJoins will naturally benefit from splitting the program into threads [Thie 10]. Those that do not feature SplitJoins can still be

parallelised by splitting a Pipeline into multiple parts. For example, benchmark *FIR* features no SplitJoins, yet splitting the Pipeline in two will result in a 1.40x speedup. However, it is important to note that whilst finding the optimal thread mapping may result in higher performance improvements than finding the optimal composition for a single-working-thread, the best performance is always obtained through a combination of both techniques. For cases such as *BeamFormer* the optimal pairing results in a 1.8x speedup compared to simply finding the best multi-threaded version. On average, the optimal combination leads to a 1.5x performance increase compared to only multi-threading.

5.3.5 Summary

This section has demonstrated that each parameter has a large effect on the performance of the workload. Regardless of using core composition, there exists an optimal number of threads for each benchmark. Unrolling allows to exposing more opportunities for composition due to increased block sizes but there is a balance to strike between extracting large blocks and TLP. Figure 5.11 shows there is a 3x benefit overall by automating the partitioning of both the software (threads) and hardware (cores).

5.4 Thread and core configurations model

As seen in the previous section, selecting the right number of threads and a good combination of cores is difficult. This difficulty arises from trying to balance between exploiting instruction level parallelism (ILP) by composing a large number of cores and exploiting TLP by generating more threads. As the design space is large, it is important to automate the decision making to alleviate the task of getting the best performance out of the streaming application at hand. The problem of automating this decision can be decomposed into two stages; first, determining the right number of threads and second, selecting a good core composition. Partitioning the program is done first as it is more natural to split the program up into threads before determining how many cores each thread requires. In this section, two machine-learning models that predict the best thread partitioning and core composition to maximise performance are presented.

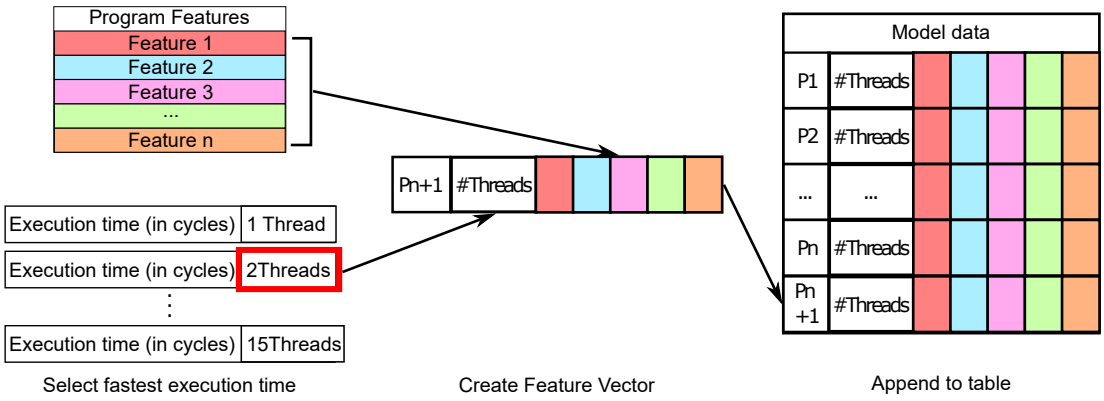


Figure 5.12: Visual representation of how data was collected for the threading model.

5.4.1 Predicting the number of threads

5.4.1.1 Correlation Analysis

To build the threading model, the synthetic benchmarks previously described in 5.2 are used. Figure 5.12 shows the overview of how the data for the model is collected. For each of the synthetic benchmarks 15 different threaded versions are generated and assigned a single core per thread. They are then all executed and their cycle count is recorded; this is repeated for 1000 unique synthetic applications. Once all the data is generated, the number of threads that leads to the fastest execution time is saved to a table with a set of features that define the program.

In order to build the two machine learning models an initial set of over 50 features are extracted from StreamIt programs. These features are extracted using pre-existing analytical tools within StreamIt and some counters added specifically for this chapter. As some of the 50 features may not contain any valuable information, the features selected for the models are determined through correlation analysis. The highest correlating features are used by the model to make a prediction about the number of threads to use. Figure 5.13 shows the 10 variables that correlate the most with determining the optimal thread number. It is important to note that these features all correlate positively with the number of threads the program requires to get the fastest execution time. In StreamIt the term *multiplicity* defines the number of times a filter will have to execute in a time slice when the graph is in a steady state [Gord 02].

The 10 features can be described as followed:

- Number of Distinct Multiplicities: the variety of different execution rates of filters per time slice.

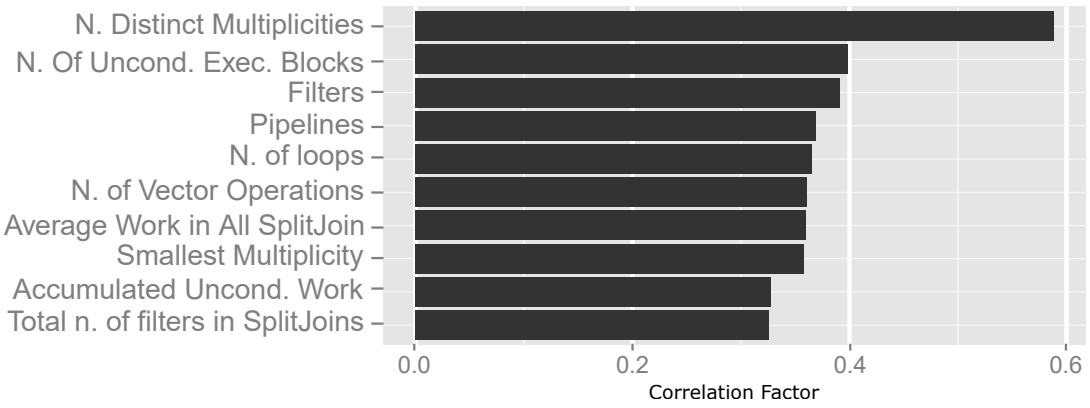


Figure 5.13: The ten highest correlating features with the best number of threads for 1000 synthetic benchmarks.

- Number of unconditionally executed blocks: number of operations in a filter that always execute.
- Filters: number of filters found in the benchmark.
- Pipelines: number of pipelines found in the benchmark.
- Number of Loops: number of loops present in the benchmark.
- Number of Vector Operations: number of potential vector operations found in the benchmark.
- Average Work in all SplitJoins: average number of operations per SplitJoin.
- Smallest Multiplicity: The smallest execution rate of a single filter.
- Accumulated Unconditional Work: Total number of operations in all filters which must be executed.
- Total Number of filters in SplitJoins: Amount of filters that are found in SplitJoins.

According to Figure 5.13 the highest correlating value is Number of Distinct Multiplicities found in the StreamIt application. There are very few variables that highly correlate beyond Number of Distinct Multiplicities. A high number of distinct multiplicities implies that the StreamIt application features an important number of filters with different execution rates. This means that certain filters may be local bottlenecks in a Pipeline for example. Multi-threading the pipelines with local bottlenecks will help alleviate the problem of multiple distinct multiplicities as it can isolate the filters with longer firing rates (filters that require multiple inputs to produce a single output) to allow smaller filters to execute more often.

For example, the benchmark *ChannelVocoder* only has 66% of its filters sharing the same average multiplicity of 50 [Thie 10], with a minimum multiplicity of 1. The

benchmark features a single SplitJoin yet when recalling section 5.3's Figure 5.11 *ChannelVocoder*'s performance is greatly improved via multi-threading. The number of threads also depends on certain structural features such as Pipelines, SplitJoins and number of Filters. Yet, these variables seem to hold less influence on the number of threads a program needs than the different multiplicities found in the graph. This is most certainly due to the fact that whilst SplitJoins make parallelisable areas more visible, if there is very little work in each stream, increasing the number of threads will make performance worse as it increases the communication to computation ratio.

It is also important to understand that a high number of Pipelines implies the use of SplitJoins. This is due to the fact that a StreamIt application with no SplitJoins will feature only a single Pipeline, thus a larger number of pipelines implies at least one SplitJoin. This is why the number of SplitJoins is not present in the correlating features; because the number of Pipeline already correlates with this feature.

A method for determining how many threads an application requires is to build a database of programs. Each entry in the database contains a feature vector for a program and the number of threads that leads to the best performance. When a new program is encountered, the database can be searched to find applications that have similar features to determine the number of threads needed. To implement such a model, k-Nearest Neighbour (kNN) is deployed. As programs will most likely not have the same values for each feature, using kNN allows to estimate the thread count by comparing the feature vector of an unknown program with a group of known programs. Given a new application, the classifier determines the k closest synthetic applications. The distance between the features is measured using the Euclidean distance for each application. Once the set of k nearest neighbours is identified, the model averages the best number of threads for each of the neighbours to make a prediction. The parameter k was determined experimentally using only the synthetic benchmarks by varying the parameter and evaluating the accuracy. A value of $k = 7$ was found to lead to the best performance. The features chosen are the ten variables displayed in Figure 5.13.

5.4.2 Predicting the size of a core composition

In Section 5.3, Figure 5.5 showed the performance of each of the 15 benchmarks when partitioning them into threads with and without core-composition. In both cases, the number of threads is often similar. The section also demonstrated that loop unrolling can improve the performance of core composition.

Since core composition is used to improve the performance of a single-working-thread, it is important to take into consideration that partitioning the software into threads facilitates the core estimation per thread. Without determining a number of threads before predicting the number of cores the core-composition model has no information as to the number of threads or the structure of each thread. In this situation, the core-composition model would either have to make its own estimates as to the thread count, or make a general prediction for a single-working-thread. Therefore predicting core-composition comes after predicting the number of threads.

5.4.2.1 Gathering Training Data for Core-Composition

For this section the single-working-threaded version of the StreamIt benchmark are used to determine the optimal number of cores in order to explore all possible core composition sizes. Some of the multi-threaded versions of benchmarks can be used to add extra data-points to build the model, however not all thread-counts are suitable. One of the difficulties of adding data-points from the highly threaded versions of an applications is that each thread will only be able to have a very small core-composition. For example, if the 15 threaded version of a benchmarks is added as data points to the model, then each of the feature vectors for this version would have a single core attributed to it. This is due to the fact that in the 15 threaded versions of benchmarks each thread can only have a single core due to the number of cores on the DMP. Yet, these threads could potentially benefit from core-composition, so adding them as data points to the model skews future predictions as the feature vector for each thread would associate the features to use only a single core. Thus high-threaded versions of applications must be ignored to avoid having incorrect suggestions for core-composition sizes.

For the exploration of core composition, the 15 StreamIt benchmarks explored throughout the design space exploration are used. To increase the amount of data available, multiple versions of the benchmarks using different amounts of unrolling are included in the search space. Four different levels of unrolling are used in to build the model: 0,4,16 and 64. To determine the optimal number of cores only the training data that has a performance within 1% of the best in the sample space is selected.

5.4.2.2 Correlation Analysis

Figure 5.14 shows the highest correlating features with the optimal number of cores, using the same set of features as the thread predictor.

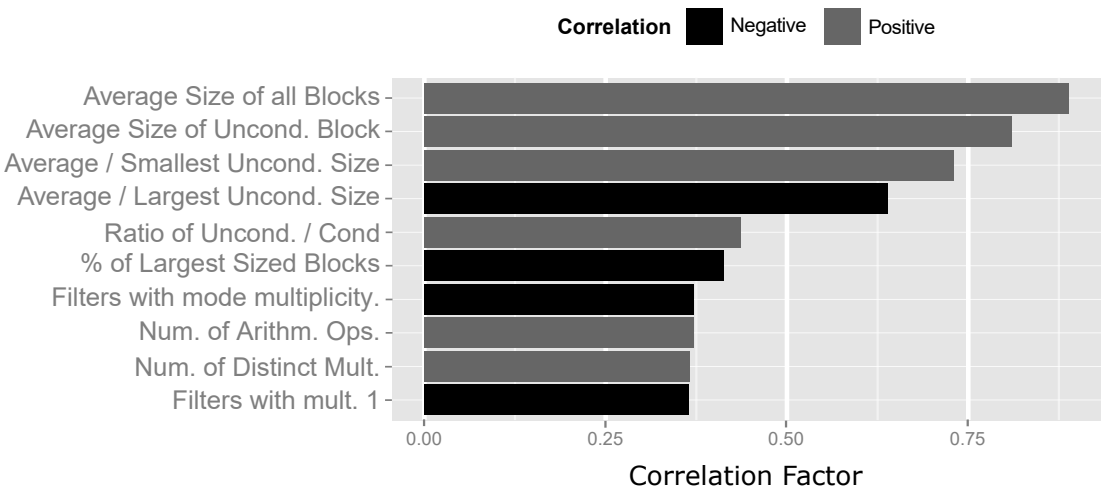


Figure 5.14: The ten highest correlating features with the optimal number of cores.

The ten features can be described as follows

- Average Size of All Blocks: Average number of operations per block of code.
- Average Size of Unconditional Blocks: Average number of operations per blocks that must execute unconditionally.
- Average / Smallest Unconditional Size: The ratio between the average size of a block compared to the smallest size of unconditional blocks.
- Average / Largest Unconditional Size: The ratio between the average size of a block compared to the largest size of unconditional blocks.
- Ratio of Unconditional Blocks to Conditional Blocks.
- Percentage of blocks that have the largest number of operations.
- Filters with mode multiplicity: number of filters that have the average firing rate.

- Number of arithmetic operations found in the program.
- Number of distinct multiplicites found in the program.
- Number of filters that have a multiplicity of 1.

The features are very different from the ones presented in Figure 5.13 and overall there are features which correlate higher with core-compositions than number of threads. The highest correlating value, which is the average size of a block, has a correlation factor of 0.88. It is important to note that the concept of an operation here is at the StreamIt level and not the architectural level. This is because the machine learning model will get information from the source-level StreamIt translation. With that in mind, the number of operations will correlate with the number of instructions found at the instruction level.

The second feature is similar to the first but only takes into account blocks that will be executed unconditionally. Blocks found in loops are excluded for this metric as there is still some form of condition for those blocks to be executed. The next two features compare the average size of an unconditional block to the largest and smallest unconditional block. The fifth feature measures the ratio of the number of unconditional blocks to conditional.

Overall the highest correlating features are not features distinct to StreamIt, such as Pipelines or SplitJoins. This is due to the fact that, from a single-threaded perspective, SplitJoins and Pipelines are less visible in terms of performance. This is especially true of SplitJoins as they will not be distributing data amongst different threads and, technically, a single-threaded StreamIt program is a long pipeline structure. It can thus be inferred that the optimal number of cores is independent of the structure of a StreamIt program. Instead determining the correct core-composition is more dependent on the amount of computation found in each program.

From Figure 5.14 the highest correlating features fit naturally under the assumptions that higher core compositions will perform better with larger blocks of operations and thus blocks of instructions. When blocks are small, a single core can fetch and execute multiple blocks in parallel; up to 4 blocks per core when blocks are smaller than 32 instructions. Cores in a composition do not fetch blocks independently; instead one of the cores in the composition will start by fetching blocks until all its lanes are used and then submits the following predicted PC to the next core in the composition. If blocks are small this means that core-compositions will have to predict a high number of blocks to fill up all its cores. Thus large blocks reduce the number of predictions required to populate all the cores with blocks, reducing the latency of fetching blocks for all cores. The necessity to correctly predict blocks to ensure that all cores are fully utilised explains why a higher number of unconditionally executed blocks compared to conditional blocks correlates positively with large core compositions.

The importance of size is also apparent as the difference between the largest block size and the average block size negatively correlates with core-composition. The ratio of unconditional and conditional blocks is considered less important than block size due to branch prediction, however having a larger number of unconditional branches is a natural fit for larger core-compositions as it reduces the dependency on high branch-prediction accuracy.

Other features that are analysed included more fine-grained data such as the types of operations that are found in the blocks of code. This involved finding ratios of

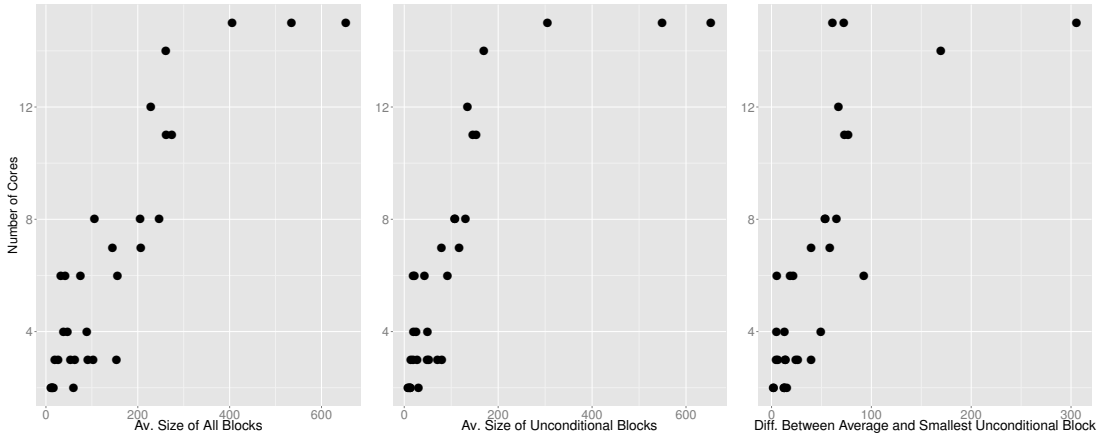


Figure 5.15: Optimal number of cores in relation to the three highest correlating features. The maximum number of cores plateaus on the right hand side as this is the maximum possible amount.

floating point, integer and memory operations. However, according to the correlation graph in Figure 5.14, the constitution of these blocks of code is not as important as their size or whether they are conditionally executed.

5.4.2.3 Linear Regression Model

Given that the optimal number of cores highly correlates with a few features, a linear regression is a natural choice to predict the best number of threads. Figures 5.15 represent how the first three highest correlating values affect the optimal number of cores for a single-working-thread. This figure is obtained by finding the best number of cores for the single-working-threaded version of each of the StreamIt benchmarks whilst varying the amount of loop unrolling. It is important to note that the reason the points in the top right corner appear to converge at 15 cores is due to the fact that no more than 15 cores can be fused. Overall, Figure 5.15 shows that StreamIt applications with large unconditionally executed blocks will require large compositions.

5.5 Results

This section describes the performance achieved by the model when predicting the number of threads and core composition to use for each of the StreamIt benchmarks and compares it to the optimal solution found when exploring the space.

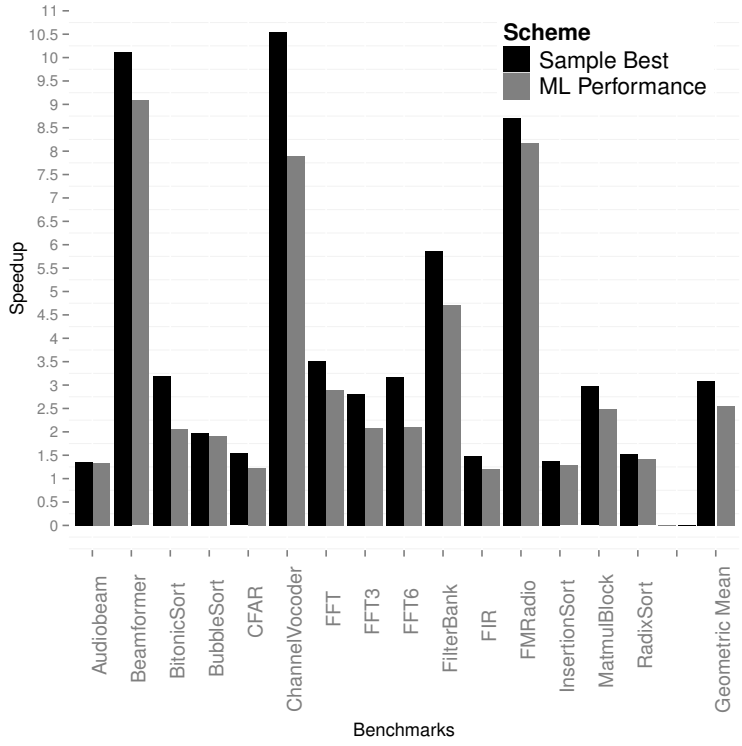


Figure 5.16: Performance of the machine-learning model against the best execution from random sampling. The baseline for the speedup measurement is single core, single-working-thread execution using O2 compiler optimisations. Higher is better.

5.5.1 Machine Learning Model Evaluation Methodology

Leave-one-out cross-validation defined in Chapter 2 Section 2.7.4 is used for testing the linear model. This is standard methodology in the machine-learning community ensuring that the training data is never used for testing. For the kNN model, the training data consists of all the generated synthetic benchmarks and it is only tested on the real StreamIt applications as they are not used for training. To obtain the speedup the performance of the machine learning based result are compared to the best from the sample space to running the StreamIt benchmark on a single core, single-working-thread, using O2 compiler optimisations.

5.5.2 Evaluation

Figure 5.16 compares the performance of the machine-learning model and the best performance from the sample space. As explained in the earlier section, the sampled best is drawn from a sample size of 1,316 combinations of core compositions and thread

partitions for each application when possible. The baseline is the original StreamIt application running with one thread and one core with O2 optimisations on the dynamic multi-core processor. The average speedup obtained through the machine-learning model is 2.6x compared to the baseline, this is only 16% smaller than the average of the best found, which is a speedup of 3.1x. These results are positive as it means the model's results are at least within 16% of the total best.

As can been seen in Figure 5.16 the largest performance penalty resides in the performance of *ChannelVocoder*. Table 5.5 presents the actual configuration found for the best sampled point and the machine-learning model prediction. Each column represents a different thread and the number in the cell represents the number of cores associated with that thread. For *ChannelVocoder* the model predicts only 8 threads rather than the optimal 13. Referring back to Figure 5.5 and Figure 5.11 from Section 5.3 *ChannelVocoder* always performs better when adding more threads rather than increasing the size of a core composition. This is the cause of the performance penalty, for *ChannelVocoder* it is more important to allocate a higher number of threads rather than compose cores. Aside from this case, the machine-learning model obtains similar speedups to the best sample.

For other applications such as *Beamformer* and *FMRadio*, the model is able to get very close to the optimal performance. Both benchmarks obtain at least an 8x speedup when using the model. This is an encouraging result as it shows that static-ahead of time partitioning of a DMP is an effective way of getting a good configuration. In fact, the difference of configurations for *FMRadio* between the best point from the space and the machine-learning model's is only of two cores which is small.

5.5.3 Summary

This section has shown that it is possible to build a machine-learning model that achieves high level of performance using source code features. In many applications, the model comes very close to the best from the sampled space, showing that the features used contain enough information to inform the model about the best decision.

	1	2	3	4	5	6	7	8	9	10
B Audiobeam	3	2								
M Audiobeam	2	3								
B Beamformer	1	4	2	4	4					
M Beamformer	6	4	4							
B BitonicSort	3	2	2	2						
M BitonicSort	1	2	2	1	2	2	2			
B BubbleSort	3	3								
M BubbleSort	2									
B CFAR	3	2								
M CFAR	2	2	1	2						
B ChannelVoc.	4	1	1	1	1	1	2	1	1	1
M ChannelVoc.	2	2	1	2	2	2	2			
B FIR	3	2								
M FIR	2	2								

	1	2	3	4	5	6
B FFT	3	3	5			
M FFT	6	5	2			
B FFT3	3	2	2			
M FFT3	3	2	3	3	3	3
B FFT6	7	8				
M FFT6	14					
B FilterBank	4	5	6			
M FilterBank	4	5				
B FMRadio	7	6				
M FMRadio	7	4				
B InsertionSort	3	2				
M InsertionSort	3					
B MatmulBlock	3	4	6	2		
M MatmulBlock	4	4				
B RadixSort	3	3				
M RadixSort	2	2				

Table 5.5: Number of Threads and Cores used for Best of Sample Space and Machine Learning Model.

5.6 Conclusion

This chapter has introduced the problem of partitioning both software and hardware for a Dynamic Multi-core Processor (DMP). Given the ability to execute up to 15 threads, a DMP can have up to 32,767 different configurations, making exhaustive search a prohibitive task. Whilst certain design choices, such as restricting the space to scenarios where only using multi-threading or homogeneous core composition are used appear as a solution, it reduces performance by up to 1.5x. Thus it is important to look at heterogeneous core-composition with multi-threading to get the best performance.

In order to quickly find a good configuration, machine learning is used and a set of 15 StreamIt benchmarks were explored. Using the early-stopping criterion, it is determined that exploring 1300 different configurations of the processor is a good representation of the performance space for each of the applications. By running each benchmark under the different configurations, each benchmark was analysed to evaluate how much performance can be obtained through the best configuration. On average, finding the optimal configuration and using the appropriate loop optimisations can lead to a performance increase of 3x compared to running the program on a single core.

Using the data from the analysis, two models were created, one that predicted the number of threads via k Nearest Neighbours, and the other predicted the number of

cores to compose per thread using linear regression. The two models are able to predict configurations close to the performance of the best design points from the sampled space; on average within 16% of the best. This demonstrated that by analysing program features, the decision of how to configure a DMP can be fully automated.

To summarise, the contributions of the chapter are:

- Proof that the configuration decision for DMPs can be automated without necessitating hand-crafted heuristics, making DMPs more practical
- An analysis of the co-design space of thread partitioning and core composition which shows how applications always perform best when using a heterogeneous configuration, leading to a 1.50x speedup compared to only multi-threading.
- A study on the impact of loop unrolling on performance, allowing for core composition to be better utilised by increasing block size, resulting in up to 3x performance increases compared to no unrolling.
- An analysis of which static code features in StreamIt can be used to determine a good configuration of a DMP.
- A demonstration that the process of determining a thread/core composition configuration that leads to good performance can be learned.
- Showing that machine learning can be applied to determine a good configuration and that the design space can be learned. The machine learning model is able to choose a configuration that results in an average performance within 16% of the best of the space.

Chapter 6

Dynamic runtime adaptation for efficient execution

The previous chapter explored how static ahead-of-time configuration of a dynamic multi-core processor (DMP) can improve the performance of multi-threaded streaming applications and showed how a mix of core and thread partitioning leads to optimal performance. Static ahead-of-time configurations can improve the overall performance of programs, however they cannot adapt themselves to exploit the different phases a program may have. A program phase can be used to describe multiple concepts, for example regions of code that exhibit different Instructions per cycle (IPC) performance, or variations of instruction mixes. When programs exhibit different phases of IPC, this can result in situations where composing multiple cores may be an inefficient solution, thus wasting resources. Chapter 5 explored multi-threaded applications yet core composition is designed to improve the performance of single threaded applications [Ipek 07] as it has become harder to improve performance for these types of application. Therefore this chapter focuses on how a DMP can reconfigure itself at runtime to exploit phases of a set of single-threaded applications.

As previously discussed the reconfiguration can happen ahead of time, especially when programs do not have a high variation in IPC. However, programs can benefit from different adaptations of the hardware at different phases of its execution. Some programs may have phases that feature high IPC whilst other phases do not perform any better on a set of composed cores. Therefore, for programs that feature many phases, runtime reconfiguration is necessary to ensure efficient use of a DMP.

Whilst optimising for speed is an important method of evaluating a DMP, the ability to reconfigure the processor allows the hardware to adapt to different performance

profiles. For example, a DMP can reconfigure itself at runtime to maximise energy efficiency, instead of only focusing on speed. Yet, whilst being able to adapt to different profiles is an attractive feature, determining when to reconfigure the processor is a challenging task for programmers. It is also difficult for compilers to determine when to compose cores without profiling information as core composition is sensitive to branch prediction and memory dependencies that may only be determined at runtime.Automating the decision of when to compose cores allows the programmer to solely focus on the software rather than decide when to modify the hardware.

The previous chapter used streaming applications to explore how partitioning a DMP can improve performance. Through partitioning, the different computation phases of the streaming programs are essentially divided amongst those threads. This chapter explores a set of single-threaded C applications, where phases cannot be isolated into different threads. A domain that features different phases of computation and is also prominent in the embedded space is image and vision recognition. Since these applications are often designed as software pipelines, they present different phases throughout their execution, and thus are an interesting case for dynamic adaptation of a DMP.

The chapter starts with an explanation of the theoretical limitations of core composition and what can be expected in terms of performance. Then the next section discusses the necessity of fine grained loop optimisations as they have a large impact on performance when composing cores. Using the San Diego Vision Benchmark Suite [Venk 09] (SD-VBS) as a use case, the chapter demonstrates that programs exhibit various phases with different amounts of IPC. This is followed by a limit study on the potential for decreasing energy consumption while maintaining performance when adapting the number of cores for each program phase. The results show that using dynamic core composition can save up to 42% on average while maintaining the same level of performance as a fixed number of cores. The issue of latency introduced by reconfiguring the system is then discussed and its influence on the impact of core composition is explored. Finally, a linear regression model that predicts the optimal number of cores per phase for reducing energy consumption while maintaining performance is generated. This practical model leads to an average of 37% saving in energy with no performance loss.

To summarise, the contributions are:

- Analysis of the limits of core composition using an analytical model.
- An in-depth comparison of static and dynamic core composition schemes on the San Diego Vision Benchmark Suite.

- Evidence that core composition has the potential to offer a large reduction in energy savings.
- An analysis of how the overhead of reconfiguring the processor can affect the potential benefits of core composition.
- A demonstration that a linear-regression based model can predict the number of cores to fuse for different program phases.

6.1 Motivation

This section motivates the use of dynamic core composition and its impact on performance and energy. It also shows that loop optimisations have a significant performance impact when fusing cores. As instructions per cycle (IPC) is a commonly used method of measuring performance, it is used throughout this chapter.

6.1.1 Dynamic Core Composition

Previous work in core composition focuses on delivering performance improvements [Ipek 07, Kim 07] and demonstrates how to predict static core fusion [Mico 16]. A static ahead of time composition fuses cores into a single composition and executes a thread on it. As evident from this prior work, core composition improves the performance of the program by maximising speed. However, as will be shown a static ahead of time core compositions may not be the perfect match for all situations.

Applications often feature different phases of IPC due to varying loop patterns. To illustrate this Figure 6.1 plots the IPC performance variation over the execution of the *Disparity* Benchmark from the San-Diego Vision Benchmark Suite (SD-VBS) [Venk 09] on core compositions of sizes 4 and 16 respectively. IPC is a natural method of evaluating the performance of a core composition as increasing the size of the composition can lead to a higher number of instructions executing per cycle. In this Figure, the x-axis tick represents 640 committed blocks (see Chapter 2 Section 2.4.1) that are committed rather than being a measure of time in cycles. This is why the high IPC phases for both compositions appear to last the same length of time even though the 16 core-composition executes the blocks faster. The reason why the number of blocks committed is used as a measurement of time is due to the fact that the number of blocks necessary to execute a program are independent of the size of a core-composition. On 4 cores, the performance oscillates between an IPC of 2 and 6 depending on the phase while on 16 cores the IPC can be as high as 16.

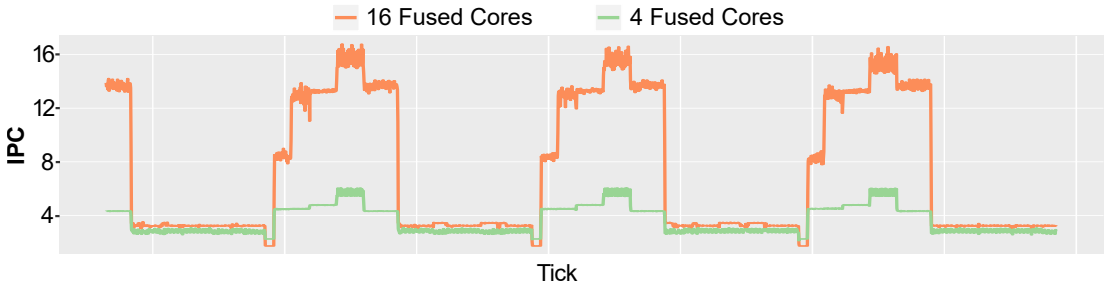


Figure 6.1: IPC of a typical benchmark (Disparity from SD-VBS) when executing on a composition of 4 or 16 core processor.

```
for(i=0; i<nr; i++) {
  for(j=0; j<nc; j++) {
    a = retSAD->data[i*retSAD->width+j];
    b = minSAD->data[i*minSAD->width+j];
    if(a<b) {
      minSAD->data[i*minSAD->width+j] = a;
      retDisp->data[i*retDisp->width+j] = level;
    }
  }
}
```

Listing 6.1: Disparity code that causes low IPC.

As can be seen in Figure 6.1, both the 4 and 16 core compositions share the same IPC when it comes to the low IPC phase. Listings 6.1 and 6.2 represent parts of the source code that lead to the low IPC behaviour and high IPC behaviour respectively. In listing 6.1, the compiler generates smaller blocks due to the control flow, which often leads to low IPC performance as will be explained later on in Section 6.2. In these situations, using a large core composition will not improve performance, which is why both the 4 and 16 core-composition share the same IPC. However the code in listing 6.2 can be unrolled and thus a high amount of IPC can be extracted. In a situation where the objective of the programmer is to maximise speed, without dynamic reconfiguration of the DMP, a static 16 core-composition will have to be used. If the static 16 core composition is used, the DMP consumes 4 times as much energy to execute the low IPC phases compared to the 4 core-composition. This is due to the fact that it uses 4 times as many resources as the 4 core composition to achieve the same performance. Thus, the static 16 core composition is considered energy inefficient during half the execution of the program.

On the other hand, if the DMP is reconfigured at runtime, the core composition can switch to 4 cores when in the low-IPC phase to save on energy and switch to the 16 core composition when in the high IPC phase to maximise speedup. In this situation, runtime reconfiguration allows to maximise speed whilst being energy efficient; a goal that cannot be achieved via static ahead of time configuration.

```
1  nr = SAD->height;
2  nc = SAD->width;
3
4  for(i=0; i<nc; i++)
5    subsref(integralImg,0,i) = subsref(SAD,0,i);
6
7  for(i=1; i<nr; i++)
8    for(j=0; j<nc; j++)
9      subsref(integralImg,i,j) = subsref(integralImg, (i-1), j) + subsref(SAD,i,j);
10 for (i = 0; i < nr; i++)
11   for (j = 1; j < nc; j++)
12     subsref(integralImg, i, j) = subsref(integralImg, i, (j - 1)) + subsref(integralImg, i, j);
```

Listing 6.2: Disparity code that leads to high IPC.

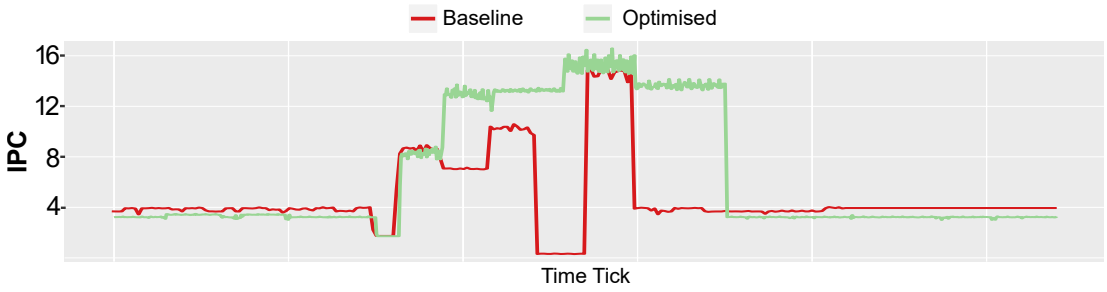


Figure 6.2: Impact of loop transformations on fused cores for the Disparity benchmark.

6.1.2 Code Optimisations

When cores are composed they execute blocks of instructions in parallel on each physical core in the core composition. In order to obtain the best performance from an application, large blocks must be generated as this leads to a higher IPC on the composition as described in Chapter 5 Section 5.3. In a nutshell, larger blocks reduce the amount of time taken to fetch blocks for all the physical cores in the composition, thus improving performance. Thus, optimisations that maximise block size will positively affect the performance of compositions. This includes optimisations such as aggressive loop unrolling, inlining and replacing conditional statements with either software predication or architecture-level predication. These optimisations are well known and do not require any structural modifications of the program.

Figure 6.2 illustrates the impact of applying loop transformations on the *Disparity* benchmark compared to a standard compiler not specifically tuned for the EDGE architecture. In this case, the two main transformations are loop unrolling and loop interchanging. The Figure shows the IPC performance of a 16 core composition with and without optimisations. For example, in Listing 6.2, the nested loop at lines 10 to 12 has a data dependency between iterations; switching the for loop headers can get rid of the dependency. As can be seen, the impact of these transformations can be large,

leading to an 12x improvement in IPC. Figure 6.2 shows that the optimisations allow the core-composition to sustain a high IPC phase longer than without the optimisations. Longer high IPC phases also means that the DMP will require less switching between core compositions, thus incurring less of a reconfiguration penalty. However, it also demonstrates that not all of the code in a program can be optimised, as the low-IPC phases do not change. More details about the loop transformations are given in section 6.4 but this example illustrates the need for careful tuning of the compiler to achieve high performance on such an architecture.

6.1.3 Knowing when and how to reconfigure the processor

Figure 6.1 motivates the use of runtime reconfiguration to ensure that DMP can improve the performance of single threaded applications efficiently by minimising energy consumption. Figure 6.2 shows how modifying loops affects the performance in terms of IPC for a core composition. Using an API, a programmer could inform the hardware when to reconfigure by using specific functions or pragmas similar to OpenMP [Dagu 98] or OpenCL [Ston 10]. Whilst this may be a viable approach to applying runtime reconfiguration, automating the decision process is a better option.

Automating the process of reconfiguring the processor enables two advantages compared to manually determining when to reconfigure the processor. The first advantage is that it removes the responsibility from the programmer: an automated runtime reconfiguration system can detect phases and adapt to them accordingly, using information gathered from previous traces. Second, if ever the program is modified, this may require new profiling information to be generated to ensure that manual reconfiguration calls are correct. If the reconfiguration is automated, then it can adapt automatically to any changes made to the source code.

Summary This section has shown that programs exhibit phases with various amount of ILP available. A DMP can take advantage of this property to fuse a large number of cores for the high-ILP phases and fuse a smaller number of cores when ILP drops to conserve energy. The section also illustrated the importance of fine-tuned code transformations to achieve sustained performance and increase the potential for fusing cores. Finally it motivates the use of automating the decision as it facilitates the utility of core composition. The next section will study in more details the expected impact of fusion using an analytical model.

6.2 A Study of Core Composition

This section studies how block size and branch prediction limit the performance of core composition. An analytical model is defined to estimate the performance of a core composition, shown in Equation 6.1. The variables in Equation 6.1 are the average block size *ABS*, the size of the composition *CS* and branch prediction cost *BPCost* which is the cycle cost of branch prediction given a branch prediction accuracy between 0 and 1. For this study, a dual issue core is used, thus the constant 2 in Equation 6.1 derives from the hardware constraint that a core can execute up to 2 instructions per cycle. *SyncCost* represents the cost (in cycles) of having to commit blocks sequentially in a composition, this is covered in more details later on. This analytical model enables a better understanding of what leads to good performance and how to determine regions of code that benefit from core composition. The limit study explains why each component of Equation 6.1 are important to predict the performance of a core composition.

$$IPC(ASB, CS, BPCost) = \left(\frac{ABS}{SyncCost(ABS, CS) + BPCost + \frac{ABS}{2}} \right) \times CS \tag{6.1}$$

$$BPCost(BranchPredAcc, BranchMisCost) = BranchMisCost * (1 - BranchPredAcc) \tag{6.2}$$

6.2.1 Branch Prediction

As discussed in Chapter 2 Section 2.4.2.2, an EDGE based DMP accelerates a single thread by executing blocks of instructions from the same thread speculatively across several composed cores. In a composition, the fetching scheme dictates that a core must fetch blocks until its instruction window is either full or cannot accommodate the newest block. Once this requirement is met, the following block is sent to the next core in the composition. If a core mis-speculates a block, this can cause the entire composition to be flushed. Thus core composition puts a strain on the branch predictor since efficiently using the cores depends on the prediction accuracy.

Depending on the sizes of the blocks and the number of cores in the composition, the branch predictor has to meet a different accuracy requirement in order to ensure that all cores are being used efficiently. In this case, efficiency is defined by cores in a composition fetching blocks from the correct branch path, meaning cores in the

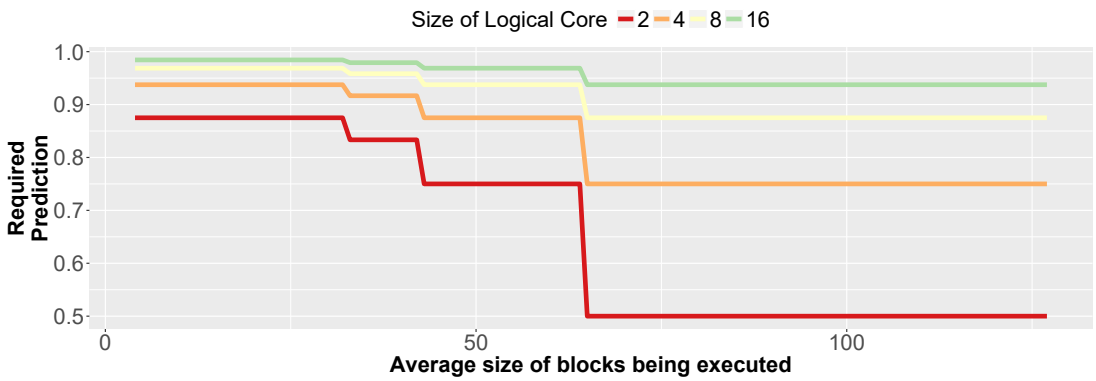


Figure 6.3: Required prediction accuracy for a logical core size to be efficient given an average block size.

composition are executing useful code. Given a composition of size *CompSize* and average block size *ABS*, the minimum branch prediction accuracy required to ensure efficient use of the composition can be determined using Equation 6.4 where *BlocksInFlight* can be calculated using equation 6.3. The groups in the Equation are derived from hardware limitations: the instruction window is divided into 4 lanes that can take a block of up to 32 instructions each. The -1 is due to the fact that when a program is executing the first block does not depend on a branch prediction, thus at any point during the execution of a program, there is a non-speculative block that does not need to predicted.

$$BlocksInFlight(AverageBlockSize) = \begin{cases} 4, & \text{if } AverageBlockSize \leq 32 \\ 3, & \text{if } 32 < AverageBlockSize \leq 64 \\ 2, & \text{if } 64 < AverageBlockSize \leq 96 \\ 1, & \text{if } 96 < AverageBlockSize \end{cases} \tag{6.3}$$

$$RequiredAccuracy(ABS, CompSize) = \frac{BlocksInFlight(ABS) \times CompSize - 1}{BlocksInFlight(ABS) \times CompSize} \tag{6.4}$$

Figure 6.3 plots the results of using Equation 6.4 with different block sizes and core composition sizes. Adding extra physical cores to a composition requires an increasingly accurate branch predictor, especially when the size of a block is under 50 instructions. This provides two insights; first of all large compositions need to run on code sections with less control flow as they are more sensitive to branch mispredictions. Second of all, branch prediction can be a simple method of evaluating the current

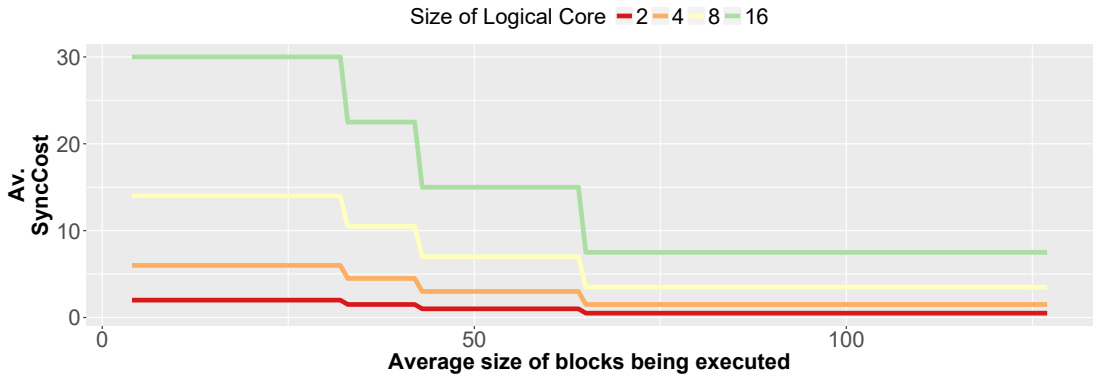


Figure 6.4: Synchronisation Cost in cycles for a given number of cores in a composition and an average block size.

effectiveness of a core composition. Given a certain number of cores, if the prediction accuracy is under the limits presented in Figure 6.3 it can be easily determined that the core composition size is sub-optimal.

6.2.2 Synchronisation Cost

For a program to execute correctly, the cores in a composition must communicate when they have finished executing a block. This ensures that the cores fetch blocks from the correct control paths and update memory and registers consistently. A core may have to wait for other cores to commit before fetching a new block. The worst-case estimate, being the highest amount of potential stalling, is defined as the **Synchronisation Cost**.

$$SyncCos(ABS, CompSize) = \frac{\sum_{CoreID=0}^{CompSize-1} (BlocksInFlight(ABS)) \times CoreID}{CompSize} \quad (6.5)$$

Blocks commit in a sequential fashion with the non-speculative block committing first and the most recent speculative block committing last. If a core's instruction window is full then it must commit a block before it fetches a new one. The Synchronisation Cost, in cycles, is defined in equation 6.5 and is measured by averaging the overall number of cycles each core in a composition waits until it can continue to fetch and execute new blocks. *AvBlocksInFlight* represents the average number of blocks in flight on a single core in the composition. When there is only a single core executing, the synchronisation cost will be 0, therefore the CoreID begins at 0. This is a worst-case estimate as block sizes will fluctuate during the execution of a program.

Figure 6.4 shows how many cycles the Synchronisation Cost will be for a given core composition size and average block size. The larger the block size the lower the

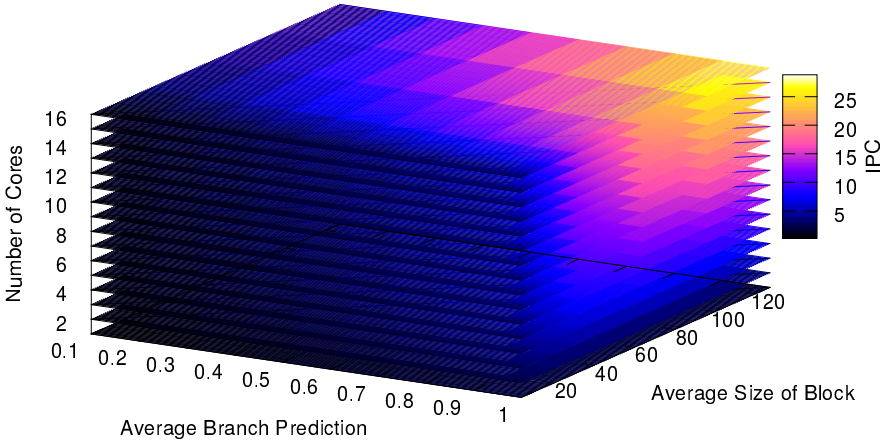


Figure 6.5: IPC estimate given a logical processor size, average branch prediction and average block size for a dual-issue core. A higher IPC means better performance.

Synchronisation Cost is since cores fetch fewer blocks and wait less for other composed cores to finish committing. Large compositions executing small blocks have a high Synchronisation Cost. This indicates that large compositions should be avoided when dealing with smaller blocks as the Synchronisation Cost outweighs the code execution.

1 Summary This section estimates the worst-case IPC for a core composition using Average Block Size, Average Branch Prediction, and Synchronisation Cost using the analytical model presented in Equation 6.1. Figure 6.5 presents a worst-case estimate of IPC performance assuming each core can sustain an IPC of 2. The figure using Equation 6.1 with different block sizes, accuracies and core composition sizes to generate estimated IPC values. Figure 6.5 shows that to obtain optimal performance requires a high branch prediction accuracy and large blocks. It shows that larger compositions can easily under-perform; for example it can be seen that 16 composed cores often have IPCs under 15, meaning that each core has an IPC under 1.

6.3 Methodology

This section now presents the experimental setup used for the remaining parts of the chapter where a thorough evaluation of core composition is conducted with the cycle-level simulator described in Chapter 4 Section 4.1.

	Disparity	Localization	MSER	Multi_NCut	Sift
Input	VGA	VGA	CIF	SIM_FAST	CIF
Cycle Count	682M	526M	175M	180M	1445M
	Stitch	SVM	Text. Synth	Tracking	
Input	CIF	CIF	FULLHD	VGA	
Cycle count	571M	577M	516M	374M	

Table 6.1: Datasets used for each of the benchmarks and the execution time (in cycles) for each of the benchmarks.

6.3.1 Benchmarks

For this chapter the performance of the Dynamic Multi-core Processor (DMP) is studied on a set of Vision Benchmarks designed for hardware and compiler research [Venk 09] described in Chapter 4 Section 4.1. The San Diego Vision Benchmark suite (SD-VBS) is composed of nine single-threaded C benchmarks ranging from image analysis to motion tracking.

Input Size The SD-VBS benchmark suite comes with a different set of input sizes. Due to executing these benchmarks on a cycle accurate simulator, executing on large datasets can take a large amount of time. A single experiment can take up to 6 hours on a single machine (Intel i5 3570k 3GHz, 16 GB of DDR3 RAM), and the average Million instructions per second (MIPS) of the simulator is 0.1 when only simulating a single core. In the paper describing each of the applications in the SD-VBS suite [Venk 09], Venkata *et al.* show that increasing the size of the input does not drastically change the phases of each benchmark. Table 6.1 shows the datasets used for the benchmarks in this chapter. For this chapter, the aim is to have a dataset that leads to executing at least 100 million cycles as this ensures that the caches and branch predictor are warmed up [Duba 13]. The execution time for each of the programs, using a single core, can be found in Table 6.1.

6.3.2 Measuring Performance and Power

The objective of the chapter is to explore how adapting the DMP to the phases of an application can affect performance. To evaluate dynamic adaptation, it is compared to different static ahead of time configurations.

Five simulations per benchmark are run, one for each core composition size: 1, 2, 4, 8 and 16. For each the IPC is recorded at an interval of 640 committed blocks. 640 committed blocks is chosen as it allows each core in a core composition to execute enough blocks before taking the measurement. This is due to the fact that the highest core composition of 16 cores can execute up to 64 blocks at a time, thus recording performance after 640 blocks allows each core to have executed at least 10 blocks. Using committed blocks allows us to easily compare each simulation as the total number of committed blocks does not change even if the core compositions are different.

The EDGE architecture is fundamentally different from the traditional CISC/RISC paradigm and thus, McPAT [Li 09] cannot be used to model power consumption as it differs from traditional CISC/RISC cores modelled in McPAT. Instead a coarse-grained power model is used where power gating is applied; when a core is not currently being used, it is assumed to be turned off and therefore does not consume energy. Using a coarse grained power model does have some limitations; mainly that when a core is active but not executing code, the model considers that it is still drawing the maximum amount of power possible. This means that this power model is pessimistic and thus the energy savings shown throughout this chapter may not fully reflect what could be achieved on a real physical processor. Nonetheless this approach still allows for an analysis of how dynamic adaptation can improve energy consumption.

6.4 Code Optimisations

This section describes optimisations focused on reducing control flow and expanding block sizes which is necessary for high performance as seen in section 6.2.

6.4.1 Loop Unrolling

As seen in Chapter 5 Section 5.3, loop unrolling can be used to reduce the overhead of the loop header and to better expose Instruction Level Parallelism (ILP). When dealing with tightly-knit loops, compositions may perform poorly due to the fact that they execute many small blocks, increasing the Synchronisation Cost and the branch prediction accuracy requirements. Unrolling loops reduces the number of blocks required to execute the loop and increases the size of the blocks, thus reducing the Synchronisation Cost and increasing ILP. For example, the innermost loop in Listing 6.3 should be completely unrolled and its outer loop unrolled to increase the block size.

```
for(int i = 0; i < 1000; i++)
  for(int j = 0; j < 1000; j++)
     for(int k = 0;k < 5; k++)
         a[i][j] = a[i][j] * b[k][j];
```

Listing 6.3: Synthetic example of an inner-most loop which should be completely unrolled.

```
for(int i = 0; i < 1000; i++)
  for(int j = 0; j < 1000; j++)
      a[i][j] = a[i][j-1] * b[i][j];
```

Listing 6.4: Synthetic example of a data dependency which can be removed via loop interchange.

There are certain factors that limit the usefulness of loop unrolling. In the evaluated EDGE architecture, blocks may not have more than 32 load or store instructions as described in Chapter 2 section 2.4.1. Therefore unrolling memory intensive loops will not always drastically increase the size of a block if the block is composed mainly of load and store instructions, as the EDGE compiler will have to split blocks if they contain more than 32 load/store instructions. However as it will generate blocks with fixed branches it reduces the strain on branch prediction. Another issue is that unrolling loops with conditional statements may not help improve the size of the block as the conditional branches might still segment the new blocks. So these loops should not be unrolled as they will lead to an increase in code size.

6.4.2 Loop Interchange

When dealing with nested loops there is one reason for interchanging the loops. The case arises when interchanging the loop removes dependencies in the inner-most loop. For instance, the dependency in listing 6.4 can be removed by interchanging the loops. This allows the compiler to unroll the inner loop efficiently, but also remove any kind of dependency between blocks. Even if memory dependencies can be detected using a dependence predictor [Chry 98], it can potentially serialise memory operations when multiple iterations of a loop are live. Thus using loop interchange will reduce any potential data dependency and minimise core communication in a composition.

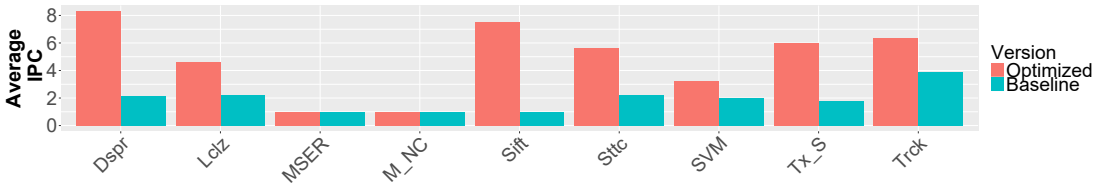


Figure 6.6: Average IPC using the optimal sized composition, with and without optimisations. Higher is better.

6.4.3 Predication and Hyperblock Formation

EDGE compilers must split blocks whenever control-flow is present [Smit 06a] as seen in Section 2.4 of Chapter 2. If a loop contains a conditional statement, the loop body has to be split in two unless if-conversion is applied. Hyperblock formation aims to reduce branching and increase block size by combining two or more blocks into a single predicated block [Smit 06a]. Hyperblocks reduce both synchronisation cost and branch prediction requirements as discussed previously. This is especially important in control-flow intensive loops where unrolling increases the number of conditional statements. Hyperblock formation can be done automatically [Smit 06a] via a compiler flag provided by the EDGE compiler. However, as of the writing of this thesis, a block can only support a single predication, thus hyperblock formation cannot be paired with loop unrolling for example.

6.4.4 Optimisation Methodology

While the optimisations described above and their tuning would be easy to implement in a compiler, this thesis did not have access to the compiler's source code. Therefore the source code of the benchmarks is modified by manually interchanging or unrolling loops. In the case of predication and hyperblock formation, simple if-then-else statements are converted into ternary operators whenever possible. Statements are also re-ordered within the body of a loop to avoid having control flow in the middle of the body. For loop unrolling, the loops were unrolled to fit a single block. The source code modifications are then verified to have the intended effect by disassembling the binary file produced by the compiler. On average there are between 0 and 12 loops modifications depending on the benchmark.

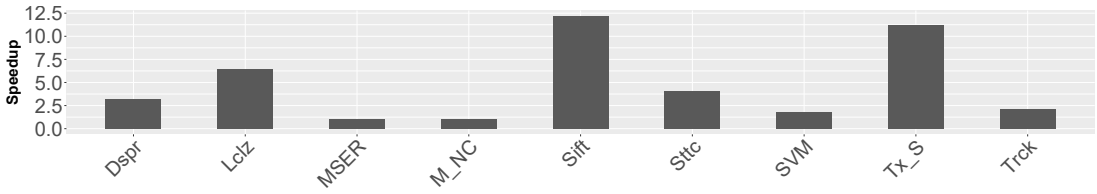


Figure 6.7: Speedup from using code-optimisations over baseline source code using the same optimal sized composition.

6.4.5 Results

First the best static core composition using the optimised code is compared with the unmodified code, both version compiled with `-O2` which is the highest optimisation setting for the EDGE compiler. Figure 6.6 shows the resulting IPC for the baseline case and the optimised benchmarks when run on a core with the optimal number of composed core to maximise performance. The IPC of the baseline is very low for the majority of the benchmarks which might give the impression that core composition is rather inefficient. However, after applying the simple optimisations described above, the average IPC is significantly increased in many cases.

Since optimisations change the total number of instructions, the actual speedup obtained using cycle count is also shown in Figure 6.7. As can be seen, benchmarks *MSER* and *Multi-NCut* do not perform any differently. This is due to the fact that none of these optimisations can be successfully applied on these benchmarks, as the loops cannot be interchanged or unrolled effectively, these applications are discussed in greater detail in Section 6.5. For the other benchmarks there is a significant improvements of up to 12× for *Sift* when the optimisations are applied. It is important to note that, whilst often the increase in IPC correlates with the speedup, this is not always the case. In these situations, this is due to the fact that some of the optimisations also reduced the amount of computation required to complete the program; this is the case for *Localization*, *Sift* and *TextureSynthesis*. These benchmarks benefit from other source code modifications such as forced inlining for *Localization* and *TextureSynthesis* and loop invariant code motion for *Sift* which were also applied manually.

Summary Overall, this section shows that classical loop transformations can have a large impact on the performance of composed cores. Without these optimisations, it would be more difficult to motivate the use of core fusion even at a static-level as the IPC does not deviate enough from a single core.

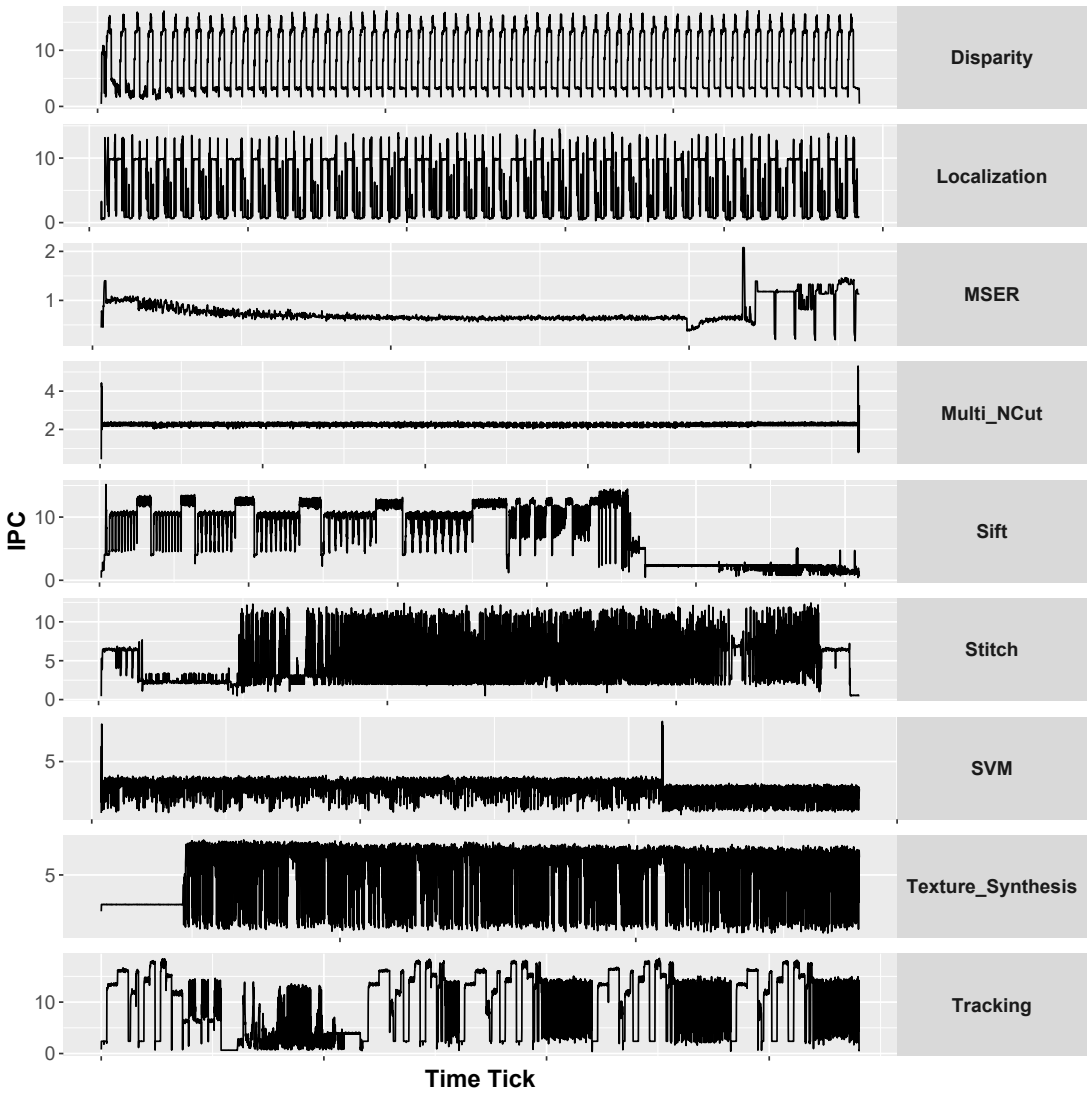


Figure 6.8: IPC as a function of time for each benchmark when run on 16 fused cores.

6.5 Benchmark Exploration

This section explores how core composition affects the performance of the SD-VBS benchmarks. First a phase analysis is performed, followed by a study of the IPC variation for static core composition. Then the use of dynamic core composition is motivated by using the gathered information.

6.5.1 Phase Detection

Figure 6.8 presents the IPC performance through time for all the benchmarks when using a core composition of 16 cores. The IPC is calculated for each time tick, which

is set at interval of 640 blocks committed. To re-iterate, the interval of 640 blocks was chosen as the largest core composition of size 16 can potentially commit up to 64 blocks at a time. Thus measuring performance every 640 blocks allows to fully capture the performance of large compositions without sacrificing loss of information.

Displaying the performance of the core composition of size 16 gives a performance ceiling for all the benchmarks as it is the maximum number of cores that can be composed at any point. Figure 6.8 shows that the IPC varies a lot for some of the benchmarks such as *Disparity* or *Localization* where dynamic composition is expected to be especially good. For other, such as *Multi_NCut*, the execution is dominated by a single long phase with constant IPC, which will clearly show no benefit from using dynamic composition.

To better understand how dynamic core composition improves performance, either through speedup or energy reductions, this section studies how the benchmarks feature different phases during their execution. Determining phases in a program requires profiling applications and analysing the profile data. This can be done with tools such as SimPoint [Pere 03] which tracks basic blocks to generate basic block vectors. These basic block vectors are then analysed using a clustering technique to determine phases.

In this chapter the objective is to reconfigure the DMP to match the different IPC phases of a program. Due to the fact that the application are already being traced for their IPC, SimPoint does not have to be applied here. For every application the IPC results of 16,8,4,2,1 fused cores are regrouped by tick and kMeans clustering [Kanu 02] is applied to detect how many IPC values can be clustered into similar groups. kMeans clustering is also used in SimPoint and more details on how it works can be found in Chapter 2 Section 5.4. Intervals that exhibit similar IPC values when run on different core counts are classified in the same cluster.

In order to find the correct number of clusters the Sum of Square Errors (SSE) is plotted for a given cluster size from 1 to 15 seen in Figure 6.9 and the optimal cluster is defined as the elbow in the plot [Ever 01]. Applying a kMeans clustering to the IPCs of each application to determine phases is preferred to determining phases of an application by reading the source code phases. This is because different source code passes may have the same IPC performance, and thus will ultimately belong to the same phase when considering core composition. For example, the benchmark *MSER*'s source code has 7 distinct passes, yet the kMeans clustering shows some of these passes result in the same performance, which can be visualised in Figure 6.8. This kMeans clustering process is only done for the purpose of exploring this set of benchmarks.

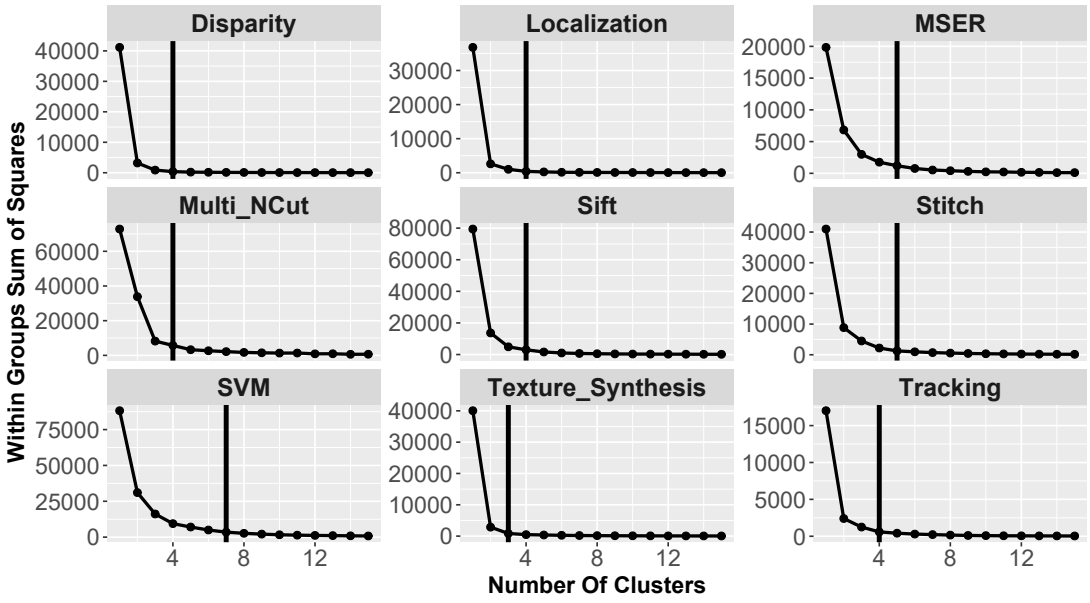


Figure 6.9: Sum of Square Errors given a number of k-means clusters for each of the benchmarks. The vertical line represents the number of clusters chosen for phase analysis.

Figure 6.10 shows the number of clusters for each benchmark and the number of times a tick in each cluster occurs throughout the execution of the program and the average IPC observed for each cluster. The frequency of a cluster is counted by how many ticks in the program are part of that specific phase. The data can be corroborated with the information found in Figure 6.8. For example, benchmark *Multi_NCut* features one phase covering over 80% of the total execution, whereas for *MSER*, the two dominant phases have very similar IPC values. This means that it will be impossible to obtain any kind of performance improvements through dynamic reconfiguration due to the fact that there is little opportunity to switch the size of the composition. For all the other benchmarks, they each have at least two dominant phases. Since each phase is a cluster of similar IPC values, having two or more clusters will result in a higher chance of benefiting from dynamic core composition.

6.5.2 Static Ahead-of-Time Core composition Exploration

In this chapter, Static Core composition defines a processor configuration that is determined ahead of time and does not change once it has been set. Figure 6.11 shows how the average Instructions Per Cycle (IPC) changes as the size of a core composition is increased, going in powers of 2 from 1 to 16 fused cores. It can be seen that, for most

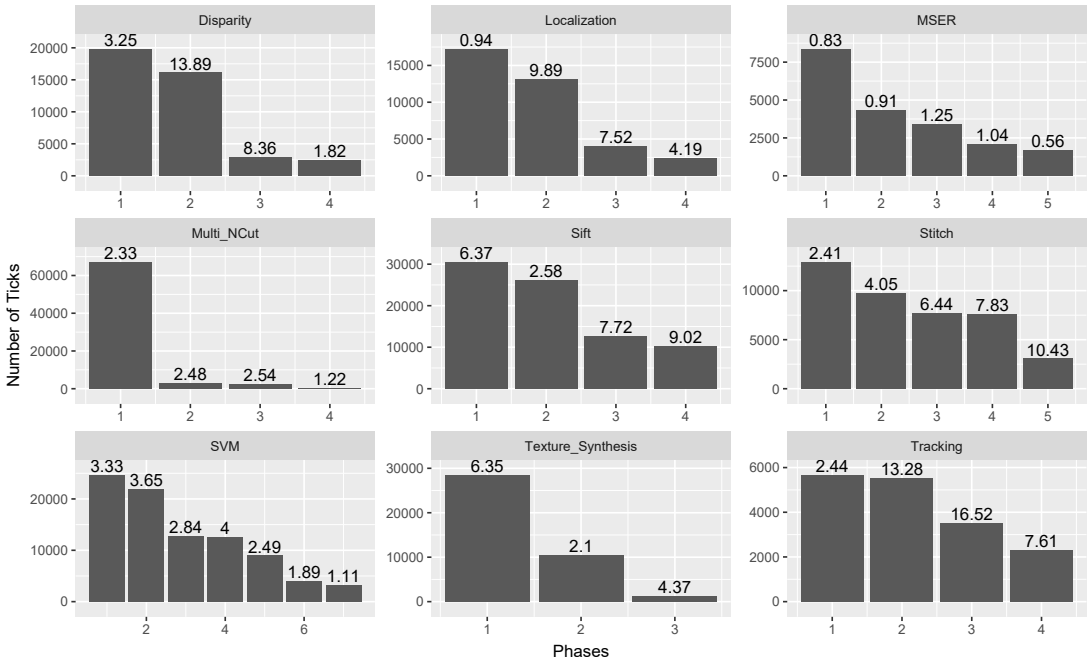


Figure 6.10: Number of phases determined for each benchmark using kMeans clustering and their distribution. The number above each phase represents the average observed IPC.

benchmarks, fusing more cores provides an increase in IPC performance. Benchmarks *Disparity*, *Localization*, *Sift*, *Stitch*, *Texture Synthesis* and *Tracking* all at least observe a speedup of 2x when using core composition.

However increasing the size of a composition is not always beneficial as can be seen in benchmarks *Localization*, *MSER*, *Multi_NCut*, *Stich*, and *SVM*. For benchmarks *Localization* and *Stitch* the performance increases when fusing up to 8 cores, whereas *MSER* and *Multi_NCut* never benefit from core composition. These benchmarks may not see a reduction in performance on 16 cores due to the increase in network traffic, which will make the largest core composition size slower than 8 cores. To summarise, when cores are fused, register files, caches and LSQs are shared, which means that cores must communicate via the network to send register or memory requests to other cores when the data they required is not stored locally. Thus, having a higher number of cores composed will put pressure on the network. Referring back to Figures 6.8 and 6.10, *MSER* and *Multi_NCut* feature one dominating long phase with a very low IPC, and thus fusing cores cannot improve performance.

Figure 6.11 also shows the variation of the IPC for each given composition size represented by the greyed out areas. For example, running the *Disparity* benchmark on a composition of 16 cores, it can be observed that an average IPC of 8.3. The variation

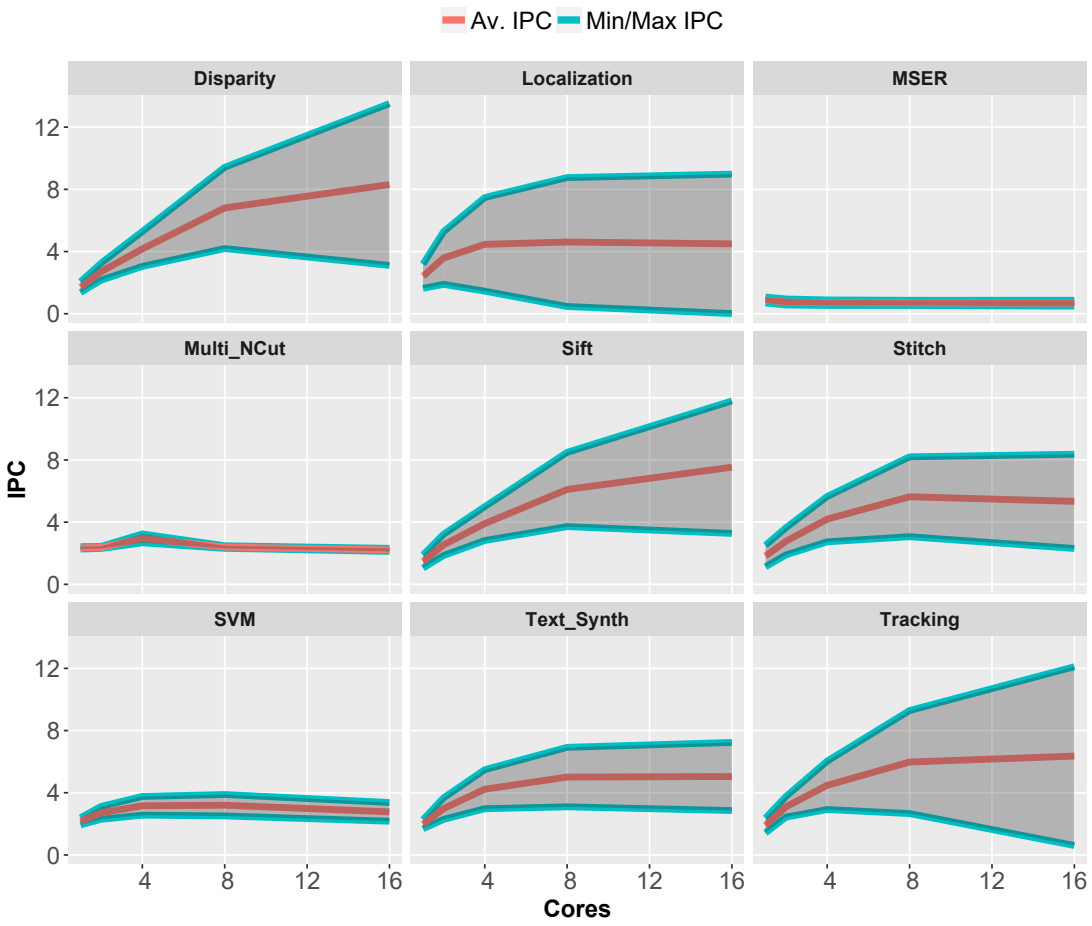


Figure 6.11: Comparing average, minimum and maximum IPC for each SD-VBS benchmark using composition size of 16.

for 16 cores shows that the performance can drop down to 2.5. An IPC of 2.5 when using 16 cores is very inefficient as this represents 0.1 of an instruction per cycle for each core and this performance can be achieved by using smaller core compositions. Using a composition of size 4 for the *Disparity* benchmark an average of 4.1 with a standard deviation of 1.2 is achieved. Thus, if the composition could change size, there is a possibility that this could reduce the overall energy consumption of the system by switching from 16 to 4.

Figure 6.12 shows the distribution of block-sizes for each of the benchmarks. As two blocks cannot occupy the same lane, which is a segment of the instruction window (see Chapter 2), the block sizes are clustered into groups which represent however many lanes will be occupied (one lane may execute a block of up to 32 instructions). Using the information discussed in Section 6.2, the size of blocks is a determining factor of whether or not a program may benefit from core composition.

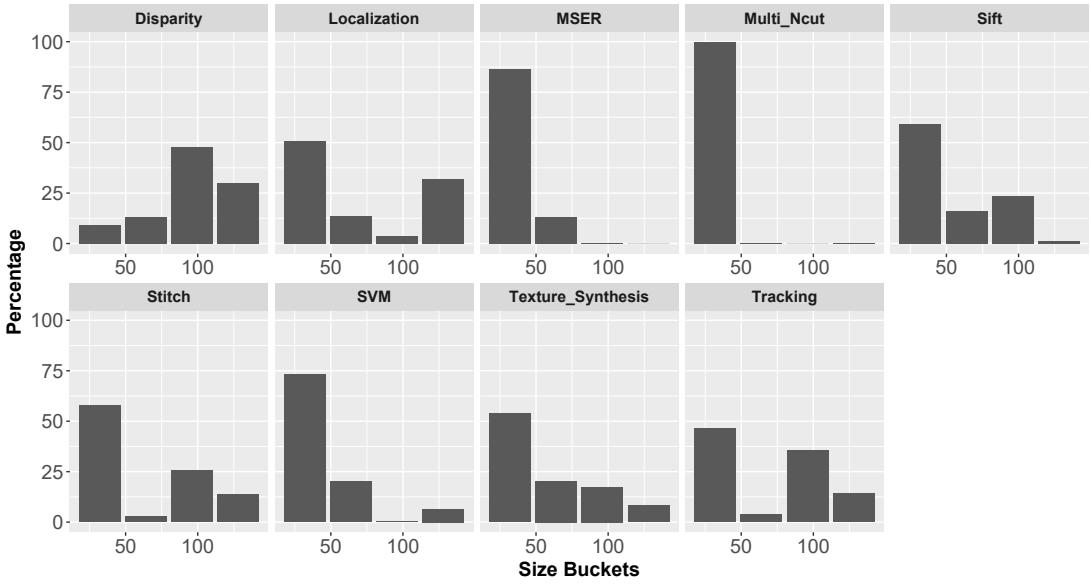


Figure 6.12: Distribution of block sizes for each benchmark. The sizes are clustered in buckets equivalent to number of lanes occupied.

Figure 6.12 helps explain why benchmarks *MSER* and *Multi_NCut* do not perform any better with core-composition. For both cases, not only do they have a single detected phase, but both are predominantly formed of blocks that will occupy a single lane. In fact, the most frequent block in *MSER*, comprising 21% the total executed blocks are comprised of only 8 instructions, whilst 31% of *Multi_NCut*'s blocks are of 28 instructions. Having such small blocks will always increase both the *SynchronisationCost* and the branch-prediction requirements.

For *MSER*, the fact that an important percentage of blocks are only 8 instructions long signifies that the overhead of fetching enough blocks for a large core-composition and the synchronisation cost for committing them outweighs executing the blocks on a single core. The reason there is no degradation of performance is due to the fact that when fusing a high number of cores, if the overhead of fetching the blocks outweighs the time required to execute then a core will not submit a block address to be fetched by another core in the composition. In this case, multiple cores may be composed, but only a single one is executing any code. For example, fusing 16 cores and executing *MSER* may in fact result in a single core being used due to it executing blocks faster than it being able to dispatch the blocks to a next core. Hence, in this case, *MSER* would be wasting a lot of energy trying to use 16 cores in a composition, whilst effectively only executing on a single core. This explains the lack of scaling for these two benchmarks. The source code of *MSER* and *Multi_NCut* is explored later on, to

Disparity	Localization	MSER	Multi_NCut	Sift
98%	95%	85%	100%	99%
Stitch	SVM	Text. Synth	Tracking	
95%	93%	98%	98 %	

Table 6.2: Branch prediction accuracy in percentage for each of the benchmarks.

demonstrate how such small blocks are generated and why it cannot be improved.

Overall, most benchmarks that benefit from large compositions will also be met with an important difference in IPC performance. The high difference in IPC performance is evidence of performance phases found in each application which are likely to benefit from dynamic adaptation.

6.5.3 Analysis of MSER and Multi_NCut

As previously mentioned, both *MSER* and *Multi_NCut* do not benefit from core-composition during their predominant phases. This subsection focuses on understanding why this is the case and explores the code that represent the main phases of both of these applications to underline where the problems arise.

MSER *MSER* has five IPC phases according to Figure 6.10; the two dominant phases represent the computation of the extremal regions tree. As seen in Figure 6.12, the majority of the blocks in MSER are 8 instructions long, meaning that core-compositions will have to load the maximum amount of blocks (4 in this thesis) to fill each core. The majority of *MSER* is composed of tightly knit loops. For example Listing 6.5 shows two loops found in *MSER's* main phase. The rest of the source code of this phase can be found in the Appendix with Listing B.1. The while loops from the listing, alongside other while loops found in the phase (lines 36-39, 40-42 and 71-75) never run for more than a few iterations. Due to the fact that these loops have small bodies, the fact that they do not run for many iterations would not make them candidates for unrolling.

It's also important to note that the average branch-prediction accuracy of *MSER* is 85%, as shown in Table 6.2. Recalling Figure 6.3 and the fact that the average block size of *MSER* is under 32 instructions, this means that no composition can be efficiently used. Some speedup can still be obtained, as cores in a composition do not need to have the instruction window full to be able to obtain performance improvements. However these improvements are minor, and come at a high energy cost.

```
27 while( forest_pt[nrindex].shortcut != nrindex ) {
28   sref(visited_pt, nvisited++) = nrindex ;
29   nrindex = forest_pt[nrindex].shortcut ;
30 }
31 while( nvisited-- ) {
32   forest_pt [ sref(visited_pt,nvisited) ] .shortcut = nrindex ;
33 }
```

Listing 6.5: Example of MSER loops (lines 27-30, 31-33).

```
17 for(k=0; k<rows; k++) {
18         for(i=0; i<cols; i++) {
19             float localMax = subsref(in,k,i);
20             int localIndex = i;
21             subsref(ind,k,i) = i;
22             for(j=0; j<cols; j++) {
23                 if(localMax < subsref(in,k,j)) {
24                     subsref(ind,k,i) = j;
25                     localMax = subsref(in,k,j);
26                     localIndex = j;
27                 }
28             }
29             subsref(in,k,localIndex) = 0;
30         }
31 }
```

Listing 6.6: Main loop of Multi_NCut.

To get performance out of *MSER*, this major phase would have to be re-written to use fewer while loops with complex termination conditions so that larger blocks can be extracted. Even with compiler optimisations, due to the fact that the loops are never executing for a long time, and their bodies are small and hard to unroll, it will be difficult to get performance improvements using core composition.

Multi-NCut *Multi_NCut* is faced with a similar situation to *MSER*. Figure 6.12 shows that once again, the majority of the blocks are less than 32 instructions long, in fact, over 50% of the blocks are of 12 instructions or less. Table 6.2 shows that the branch prediction is always correct, yet, with blocks this small, the synchronisation cost of executing the blocks on large compositions outdoes any benefit that can be obtained via core-composition. Listing 6.6 shows the main loop of *Multi_NCut*; the rest of the source code can be found in Listing B.2.Once again, this loop cannot be efficiently unrolled as the if statement can only be turned into a single hyperblock once. For *Multi_NCut* the problem is that the blocks are too small which will cause the overhead of dispatching multiple blocks across a core-composition inefficient. This is why the benchmark does not perform significantly better with core-composition without a high increase in energy consumption.

Overall, both benchmarks demonstrate that some loops cannot be adapted to benefit from core-composition. This is especially true of while loops that cannot be converted into more traditional for loops. Due to the small size of these loops, the overhead of dispatching multiple blocks over multiple cores outweighs the performance benefits.

6.6 Dynamic Core Composition

This section discusses a dynamic core composition scheme that is created by generating traces of the ahead-of-time static compositions. The section first describe how the traces are generated for the dynamic core composition schemes. Before the analysis two types of static core composition are defined:

- **Static Benchmark** (SB): A fixed fused-core which is optimal for a unique benchmark.
- **Static Suite** (SS): A fixed fused-core which represents the average best for the entire suite of benchmarks. This represents the baseline for the chapter.

The reason **Static Suite** is used as a baseline is due to the fact that it represents a configuration choice at design time. It is the equivalent of hardware designers analysing the requirements of a processor based on the type of applications it will be executing. This is better than using a single core as a baseline, as a single core will always represent the slowest execution time whilst also consuming the least amount of energy. **Static Benchmark** on the other hand represents the ability to compose cores statically ahead of time; which is an extra step of flexibility compared to **Static Suite**.

The static core-composition scheme *SS* is compared to the results obtained for the dynamic one for the SD-VBS benchmarks. This is followed by a closer analysis of the dynamic core composition objective: optimising speed whilst reducing energy consumption.

6.6.1 Creating Dynamic Core Composition Traces

Dynamic core composition enables the ability to change the number of cores for each time tick (an interval of 640 committed blocks) during the execution of a program. In order to explore the different performance and energy trade-offs that are possible to achieve with this technique, traces of execution for the application are collected. Figure 6.13 is a visual overview of how dynamic traces are collected. First the application

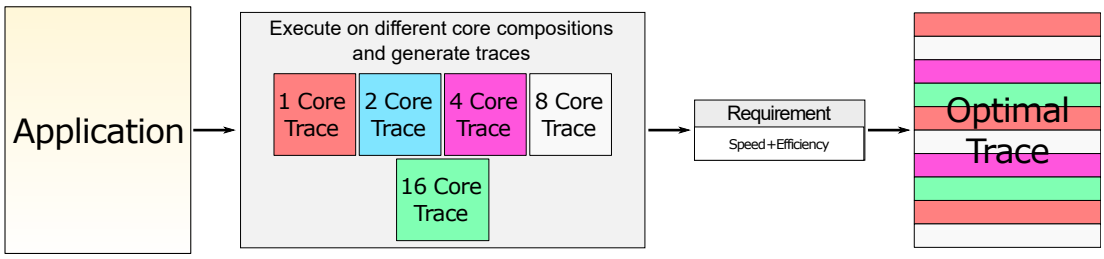


Figure 6.13: Overview of how traces are gathered to generate dynamic core composition traces.

is executed on 1,2,4,8 and 16 composed cores and the performance is recorded for each time tick of 640 committed blocks. Using these 5 traces, dynamic executions using any of the 5 different compositions can be constructed to generate dynamic traces. For this chapter, the dynamic trace are generated based on maximising speed whilst minimising energy consumption.

To simplify the exploration process, time ticks that are attributed to the same phase are always given the same number of cores. This is done to reduce the search space as on average there are 48,494 ticks which results in an average of $5^{48,494}$ different possible executions. Since the maximum number of clusters found is 7 (for SVM), a maximum of $5^7 = 78,125$ different dynamic execution traces can be built. This is why creating dynamic core-composition out of traces is preferred to running all potential dynamic configurations via the simulator. All applications run for a couple hundred million cycles, which often takes a couple of hours to execute. As the amount of dynamic reconfigurations is high for some benchmarks, using traces to simulate dynamic reconfiguration is a very large time-save.

When the size of the core composition is switched, the performance of that composition from its respective trace file is used and an extra 100 cycle penalty is added for switching the size. This 100 cycles is the overhead of reconfiguring the processor at runtime, the effect of the reconfiguration latency is discussed in further detail later on in this section. The reconfiguration is a lightweight process as described in Chapter 2 section 2.4.2 that involves informing the cores that they belong to a composition, which requires a write to a system register and potentially flushing pipelines if the cores are executing other threads. As in this chapter, cores will never be executing other threads, flushing is not necessary, thus the l00 cycles to inform cores is an appropriate penalty and has been used in previous studies on DMPs [Pric 12]. With all these different dynamic core composition traces, the optimal schemes for efficiently maximising speed can be found.

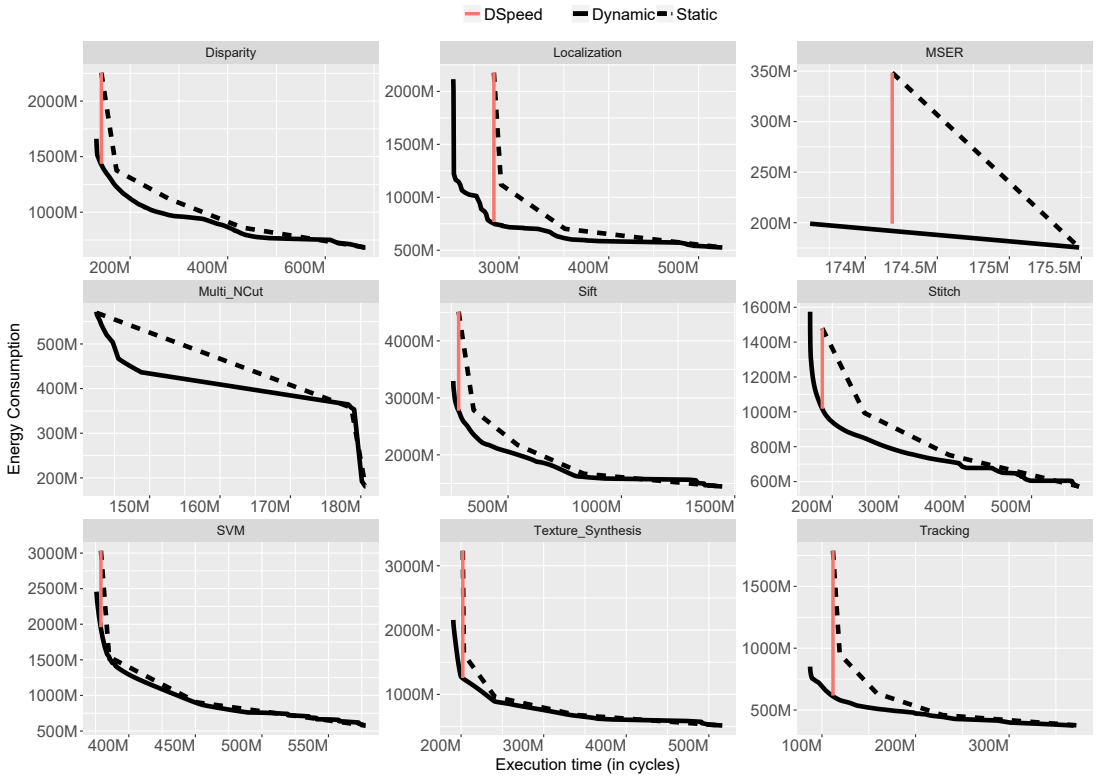


Figure 6.14: Time (x-axis) vs. Energy (y-axis) trade-offs using Static and Dynamic Composition Schemes.

6.6.2 Dynamic Core Composition

Figure 6.14 shows the trade off between time (cycles) and energy using either a static ahead of time or dynamic (re)-configuration. The dotted line represents the static core composition scheme for the benchmark whilst the solid line represents the Pareto Front of all the dynamic core composition traces. The vertical line represents the amount of energy that can be saved from using a dynamic core composition scheme that matches the same speed as the best static scheme. The Pareto Front is constructed by assigning different core composition sizes to a phase and recording the execution time in cycles and energy consumption. For a given cycle count, the reconfiguration scheme that leads to the smallest energy consumption is chosen to be a point in the Pareto Front.

Figure 6.14 demonstrates how static core-composition fails to maintain good energy efficiency as speed is improved. For example, *Disparity* is fastest on 16 fused cores, but has an 1.63x increase in energy consumption for a 1.22x improvement in speed. This is due to the fact that larger core compositions do not result in linear speedups, and thus consume more energy than a smaller core composition. When us-

ing the dynamic scheme, it is clear that energy consumption increases at a slower rate when increasing speed. In this case the number of cores is adapted to the current phase, using just enough cores to maintain high performance without wasting energy.

Figure 6.14 also shows how very few applications get faster execution times with dynamic core composition. The main program that does perform better execution wise with dynamic core composition is *Localization*. In the case of *Localization*, the fastest execution time using static core-composition comes from a logical core of size 8. However, there are certain phases that perform better with 16 cores, and thus, dynamic core composition in this situation can lead to faster execution times. Overall, for these benchmarks, most have their fastest execution times with a static core-composition. Dynamic reconfiguration is therefore mainly used to limit the energy consumption.

6.6.3 Optimising for Speed

In this section the dynamic scheme is defined to be one that matches the same speed performance as the fastest static core composition for the benchmark: **DSpeed**. This is equivalent to the vertical line found in Figure 6.14. This scheme is used to demonstrate how dynamic reconfiguration can achieve the same speed as the static configuration whilst reducing energy consumption.

Figure 6.15 shows the speedup of **DSpeed** and Static Best (SB) and the respective energy consumption. The results are normalised against the performance of Static Suite (SS), which is 8 cores fused. The SS core count is obtained by averaging the number of cores that leads to the fastest execution time for each benchmark. The execution times for **DSpeed** and SB are the same as the dynamic scheme designed to match the static best's execution time. As can be seen some benchmark perform better when using benchmark specific core compositions rather than SS. Both *Disparity* and *Sift* obtain a 1.25x speedup when using the SB scheme whilst *Tracking* benefits from a 1.10x speedup. This reconfirms the concept that even static core-composition is a feature that leads to performance improvements over design time configurations.

When looking at the Energy graph of Figure 6.15, it is clear where the SS scheme fails. Benchmarks *MSER* and *Multi_NCut* feature very little improvement when using core composition, if any, therefore SS will perform very poorly when it comes to energy consumption for the benchmarks. In the case of those benchmarks, the energy consumption is over 2x less on SB than it is on SS. In this situation, SS is analogous to designing a large physical core meant to extract a lot of IPC for single-threaded per-

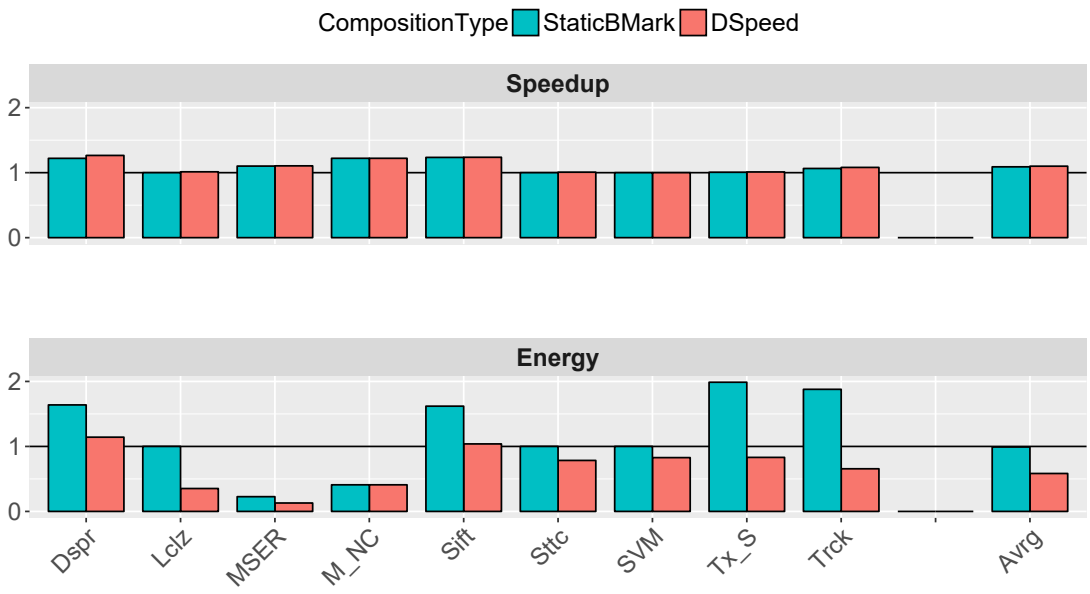


Figure 6.15: Maximising speed for all the SD-VBS benchmarks. For Speedup, higher means better, for Energy, lower is better.

formance and executing low IPC benchmarks on it. This core will end up consuming too much energy and lead to low performance increases. SB already shows the advantages of designing smaller, simpler physical cores which can be composed ahead of time. For applications such as *MSER* or *Multi_NCut*, a single low energy core suffices whereas *Disparity* and *Tracking* benefit from large compositions for better speedup.

However, SB is still not an optimal solution. For the benchmarks *Disparity*, *Sift*, *Texture_Synthesis* the energy consumption is much higher. This is due to the fact that these benchmarks perform best on a 16-core system, however as seen in Figure 6.11, the variation in performance always increases when fusing this many cores. In this situation, whilst 16 cores does result in the fastest execution times, the energy consumption is higher than SS. This shows the limitations of static configuration overall, whether it be at design time or ahead of time: the lack of flexibility means that compromises have to be made. In the case of aiming for the fastest execution times energy consumption may increase.

The dynamic reconfiguration **DSpeed** scheme always performs better than the SB scheme in terms of energy consumption and can even match the SS scheme on energy consumption whilst improving speed such as in the *Sift* benchmark. For the *Localization* benchmark, the **DSpeed** matches the performance of both the SB and SS whilst reducing energy consumption by 65%. By using **DSpeed**, the energy consumption can

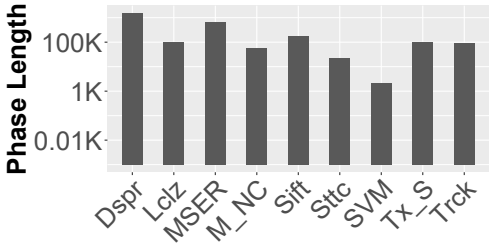


Figure 6.16: Average number of cycles without switching.

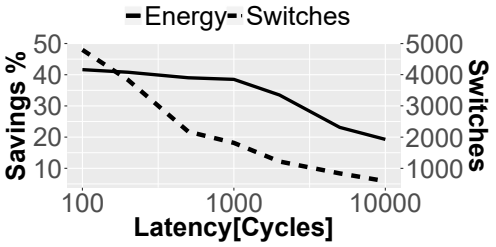


Figure 6.17: Energy savings and number of switches as a function of the switching latency in cycles.

be reduced by 42% compared to both SB and SS without impacting performance. This illustrates the greatest advantage of using a DMP since the number of composed core can be adapted continuously depending on the amount of ILP available.

6.6.4 Reconfiguration Latency

Up until now, the chapter has assumed a reconfiguration latency of 100 cycles whenever dynamic reconfiguration occurs as explained in Chapter 4. This section studies the impact of a larger reconfiguration overhead on energy savings. First, figure 6.16 shows the average phase length for each benchmark when maximising energy savings while maintaining performance (**DSpeed**). As can be seen, the majority of the benchmarks run for long period of several tens of thousands of cycles before any switching occurs on average. Therefore, even if the reconfiguration latency is increased to a larger value (e. g. 1,000 cycles), its impact might be minimal. For these applications, the phase length is also tied to the size of the input, the phases may increase when working on inputs such as high definition images.

Furthermore, there is always the option to reconfigure less often, in the case where a change in configuration only brings marginal reduction in energy. In such case it might be more beneficial to keep running on the slightly less optimal configuration than paying a cost for reconfiguration. Figure 6.17 illustrates this perfectly, showing how energy behaves as a function of the reconfiguration overhead (averaged across benchmarks). For each latency value, the best trace of reconfiguration is determined to keep performance equal to the best static configuration while minimising energy (**DSpeed**). The left y-axis expresses the energy savings relative to the static scheme, while the right y-axis shows the total number of switches. The energy savings remains high up to a latency of 1,000 cycles, with a noticeable decrease in the number of

switches. For latency values over 1,000 cycles, the energy savings drop considerably, with very little switching occurring. This data shows that even if the reconfiguration overhead is 1,000 cycles, average energy savings of 38% are possible compared to 42% when the overhead is 100 cycles.

Summary Overall, dynamic core composition will always lead to higher speedup with lower energy consumption compared to a fixed configuration. This is due to the presence of phases in applications that the dynamic scheme can exploit to reduce wasting energy in low ILP phases. This section has shown that maximising speed can be highly energy inefficient when using a static composition and that a dynamic scheme can help reduce energy consumption by 42% on average.

6.7 Linear Regression Model

The previous section showed that dynamically reconfiguring the processor can help reduce energy consumption whilst still achieving the same execution time as the fastest ahead of time configuration. In order to benefit fully from dynamic core-composition two solutions are possible; either the programmer must go through the code and manually determine when to change the composition or an automatic scheme can be deployed. This section now presents a learning scheme that is used to exploit the large energy savings available. The main idea is to monitor at runtime some performance counters and make a decision at a regular interval on how to reconfigure the cores. For this purpose, a model is trained offline using the data collected and presented earlier in the chapter. Once trained, the model predicts the optimal number of cores based on the performance counters from the previous time interval and reconfiguration occurs if it is different from the current number of cores.

6.7.1 Model

As the decisions are made at runtime, a lightweight model that is able to predict the correct configuration that can be integrated in hardware is necessary. Linear regression, which makes predictions using a weighted sum of the input feature, has been demonstrated to be useful for predicting processor performance [Jose 06]. It is chosen as it has been as it can easily be implemented in hardware [Lee 06, Luke 12] and has a low overhead when computing the sum. The model is trained offline using the traces gathered from the prior analysis for the **DSpeed** scenario which maximises energy savings while maintaining performance.

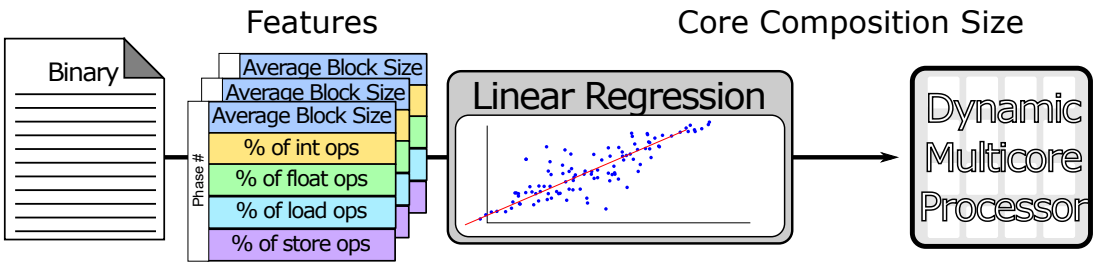


Figure 6.18: Linear Model.

Figure 6.18 shows how the linear model is trained. The dataset consists of a set of four input features (average block size, and percentage of integer, floating point and load operations) and the optimal number of cores for each time tick for each program. These features are chosen as they are easy to extract from the hardware. The reason stores are not in the feature vector is due to the fact that a block is comprised only of integer, floating point, load and store operations. Therefore, when building the model, a correlation analysis determined that stores correlate with other features and selects to remove it as a variable.

To speedup the learning process, for each benchmark the features of all the ticks in a phase were averaged out to create a single data point, which is comprised of an IPC value, and the features described in Figure 6.18. This averaging out leads to 40 data points for all the benchmarks, as this is the sum of all the phases for the suite seen in Figure 6.10. The training consists of finding the weights that minimise the error when predicting the optimal number of cores to use across all time ticks and benchmarks. Since only core configurations which use a power of two number of cores are considered, the linear model is built to predict the logarithm (base 2) of the number of cores. The prediction is rounded up to the nearest integer in the interval $[0,4]$. The following equation represents the trained linear model which can be used to make prediction:

$$log_2(\textbf{\#cores}) = -7.7 + 0.028 \cdot \textbf{avgBlkSze} + 0.075 \cdot \textbf{\%int\_ops} + 0.069 \cdot \textbf{\%fp\_ops} + 0.21 \cdot \textbf{\%ld\_ops}$$

It is important to note that this model was not used during the validation, as cross-validation (see Chapter 2) was used to evaluate it. Instead, this represents a model where the data from all programs is used. For instance, if at runtime an average block size of 6 instructions, and 77%, 1% and 18% of integer, floating point and load operations, respectively, then the predicted value will be 2.092. Rounded half up to the nearest integer value, 2, the optimal number of cores predicted will, therefore, be 4.

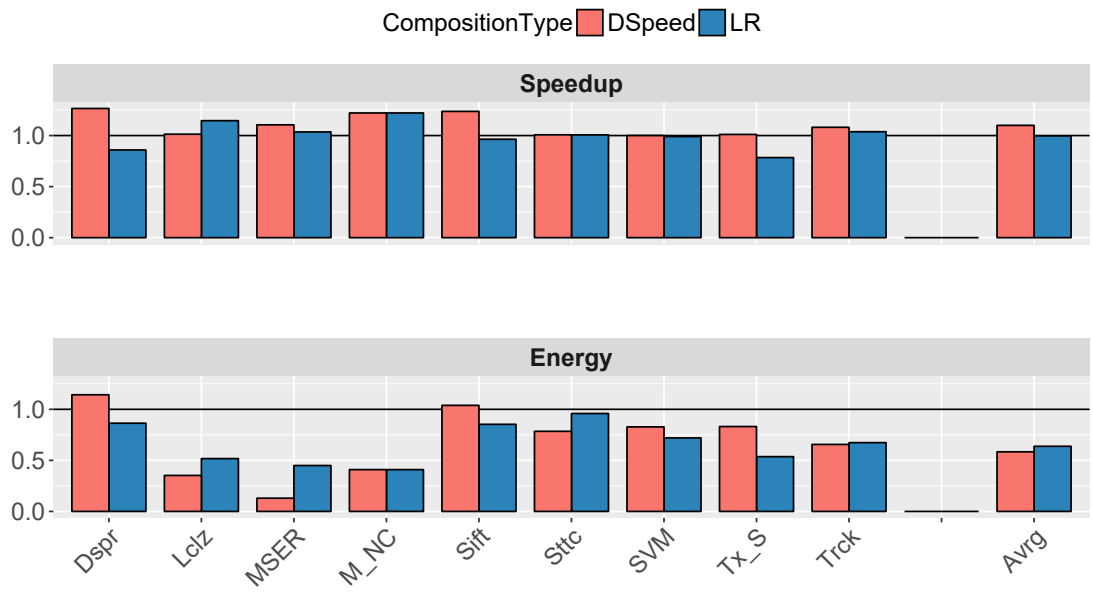


Figure 6.19: Performance results for maximising speed for the SD-VBS benchmarks using the linear regression (LR) model. The results are normalised against the Static Suite core composition.

As can be seen, the largest weight is on the percentage of loads operations. This is due to different reasons, mainly execution time and the fact that Load-Store Queues are fused. When it comes to execution time, loads may take from 3 to 128 cycles depending on whether or not it is a cache miss or hit. Whether it is a cache hit or miss, a block that takes longer to execute will often minimise the *SynchronisationCost* penalty. A block composed mainly of integer or floating point operations will often result in shorter execution rates; thus may execute faster, making it harder for large logical cores to improve performance.

More discussion on how the time it takes to execute a block influences the performance of core compositions is discussed in Chapter 7. The other reason loads have the largest weight is due to the fact that loads can be fired independently to the Load-Store Queue. Unlike stores that depend on previous memory instructions blocks being committed, loads can be fired with less overhead. As data can be speculatively fetched, load instructions can receive data from other cores before the data is stored, speeding up executions. By increasing the core count on load heavy blocks this will improve performance more reliably due to cores being able to issue loads in parallel.

6.7.2 Results

To evaluate the performance of the model leave-one-out cross-validation is used; it is a standard machine-learning methodology which tests the model using not seen during training. For instance, if the model is tested for one program, for example *Disparity*, the model is then trained using the dataset from all the other programs combined. Then the resulting trained linear model is used to predict the optimal core number for each time tick of the disparity program and report the performance achieved.

Figure 6.19 shows the performance in terms of speed and energy that is achieved using the linear model normalised by a fixed static configuration. The fixed configuration maximised performance across all the benchmarks using 8 cores and is the same as in the previous results presented in figure 6.15. On average, the linear regression model is able to consume 37% less energy compared to the 8 cores fixed configuration and is able to exactly match its speed. The main outlier is *MSER* where the linear regression consumes over 2x more energy than **DSpeed**. This is due to the fact that *MSER* is a benchmark where the branch prediction is poor as previously mentioned in Table 6.2. Indeed, *MSER* has an average branch prediction of 85% compared to the average of 95%.S As *MSER* tends to have very small blocks, the branch prediction makes it very difficult to ever efficiently use even a core composition of size 2. This benchmark is therefore an outlier compared to the rest of the set, as it is the branch prediction causing incorrect predictions from the linear regression.

The performance is also compared with the best possible choice of dynamic reconfiguration, **DSpeed** which acts as an Oracle. As can be seen, the linear model is able to exploit similar energy savings to the **DSpeed** scheme in most cases. On average it reduces energy by 37%, which is within 5 percentage points of the 42% achievable by the **Dspeed** scheme. These results show that it is possible to implement a simple realistic lightweight scheme which offers large energy savings.

6.8 Conclusion

This chapter tackled the problem of dynamic reconfiguration of a Dynamic Multicore Processor at runtime. Due to the fact that adding cores in a core composition does not result in linear improvements, obtaining the fastest performance comes at the cost of energy. Therefore using ahead of time static core compositions is not an efficient way of speeding up programs with phases as energy consumption will be high when

executing phases with low IPC. Runtime dynamic reconfiguration of DMPs is therefore necessary to ensure that core compositions are used appropriately.

To better understand how core composition is sensitive to branch prediction and block size, a limit study has been conducted. It showed that larger core compositions favour large blocks as this reduces the strain on the branch predictor and also reduces the communication cost between cores. To improve the size of blocks and block level parallelism, a set of compiler optimisations such as loop inversion, loop unrolling and predication have been discussed.

These optimisation are then applied on a set of vision benchmarks, as they are programs that feature varying phases of IPC, and the performance of static core compositions help show that these programs have phases of IPC patterns. Using this information, a dynamic runtime reconfiguration scheme is created: **DSpeed** that matches the speed of the fastest static core fusion whilst minimising energy consumption.The chapter shows that **DSpeed** saves on average 42% energy compared to the optimal static core composition core.

Finally a linear regression model has been proposed to drive the adaptation process at runtime for **DSpeed**. This model leads to a 37% saving in energy whilst maintaining the same level of performance as the optimal static scheme.

Overall, the contributions of this chapter are:

- An analysis of the limits of core composition via an analytical model that demonstrated that in order for core composition to be effective, blocks must be large to reduce the strain on branch prediction accuracy and the cost of synchronising cores.
- An in-depth comparison of static ahead of time and dynamic core composition schemes on the San Diego Vision Benchmark Suite which demonstrated that the benchmarks benefit most from dynamic core composition as it can achieve the same speedups as static ahead of time whilst reducing energy savings.
- A demonstration that core composition has the potential to offer a large savings in energy savings of up to 42% compared to the static ahead of time core composition.
- An analysis of how the cost of reconfiguring can affect the overall energy savings. The chapter showed that for the set of benchmarks, the reconfiguration cost can be as high as 1000 cycles without greatly impacting performance.

- A demonstration that a linear-regression based model can predict the number of cores to fuse for different program phases using static code features and can achieve similar energy savings as the optimal execution with 36% energy savings on average.

Chapter 7

New fetching scheme and data speculation for improved performance

7.1 Introduction

The previous chapter showed how reconfiguring a dynamic multi-core processor at runtime can improve the efficiency of core composition as it is able to adapt to different phases of instructions per cycle (IPC). It also discussed limiting factors of performance such as branch prediction requirements and cost of synchronising cores. To improve the performance of core composition, Chapters 5 and 6 showed that source level modifications are a good method. These source-level modifications are often used to increase the size of the block which enables better utilisation of large core compositions.

In situations where source or compiler optimisations cannot increase the size of a small block, core composition cannot improve the performance of the application. This is due to the latencies introduced by fetching multiple small blocks execute on a composition. To increase the viability of core composition, other solutions must therefore be explored. Instead of only focusing on improving the source code, analysing how a composition functions at a hardware level can help determine other bottlenecks.

In order to improve the performance of larger core compositions, some of the mechanisms may need to be added to the processor.For example, having to fetch multiple small blocks on large core compositions introduces a fetch latency and reduces the potential speedup. Introducing a new way of fetching blocks amongst cores can potentially reduce this latency and thus increase the efficiency of the composition.

This chapter looks at two features of the processor that have a large impact on performance: the block fetching mechanism in a composition, and data dependencies that arise between blocks. The current fetching model focuses on filling the instruction window of a single core before activating another core in the composition. Without modifications, this fetching model requires large blocks to reduce the time required to activate multiple cores in a composition. Thus, adding a new fetching model that prioritises using all the cores in the composition over filling a single core can lead to better utilisation of the composition.

Secondly register dependencies cause blocks to take longer to commit, as they will have to wait on the register to become available, reducing instruction level parallelism (ILP). Reduced ILP due to data dependencies is similar to an issue found in superscalar processors [Pera 15]. If register values could be predicted, instructions could fire speculatively improving ILP. This chapter therefore explores how a value predictor, which predicts register values to reduce the data dependencies, can be used to improve performance in core composition.

These two factors that introduce latency on large compositions have previously been discussed in the work of Robatmili *et al.* in [Roba 11] for a TFlex [Kim 07] processor. The TFlex is also an EDGE based processor (see Chapter 3 Section 3.1.1) however it lacks the ability to execute multiple blocks on a single core. In this work they propose two features to tackle these issues: selective re-issue of blocks and bypassing register writes for critical instructions.

Selective block re-issue involves using the instruction window as a buffer: blocks can remain in the window after having been committed. When a core fetches a new block PC, it checks if any of the idle cores in the composition already have the block in their window and if one does, it tells that core to start executing the block instead of submitting an I-Cache request. An idle core can be a core that has just committed a block, and thus if it is executing a loop composed of a single block the core could simply avoid re-fetching that block by refreshing it. Whilst this is an interesting approach to reducing the cost of fetching blocks on a composition, it does not address the issue of populating large compositions quickly. In the case of the processor used in this thesis, a core can fetch and execute up to four blocks which is higher than TFlex. This means that if the fetching latency is long, there's a high chance that a re-issue policy would favour populated cores in the composition that already have the next-to-be-fetched block, leaving some cores that have yet to receive a fetch request empty. Re-issuing blocks may reduce the fetching latency on the sub-set of active cores, how-

ever this will not allow the composition to be used at its fullest. Thus another fetching policy that focuses on ensuring each core is active to fully use the composition must be designed.

As for register bypassing, they propose a distributed block criticality analyser (DBCA) that can detect instructions that cause inter-block data dependencies. Once these instructions are detected, whenever they finish executing, their values bypass the register file to be sent only to the successive speculative block that depends on the value. Whilst this technique does reduce some of the latency caused by inter-block data dependencies it still requires that the data-dependent block wait for the instruction that generates the value. If multiple blocks are fetched and executed in parallel, and the bypassing is only done between two blocks [Roba 11], then there is still going to be a latency caused by multiple blocks having to produce an output before forwarding the value. With value prediction, cores can speculatively fetch register values without having to communicate, allowing for a greater amount of ILP to be exploited.

This chapter is organised as follows: first a benchmark previously explored in Chapter 6 is re-analysed to underline how current hardware does not suffice to ensure performance improvements. Then a new fetching mechanism called Round Robin Fetch is introduced: cores are able to fetch blocks independently in a round robin fashion to improve the distribution of work amongst the cores in the composition. Then a current state-of-the art value predictor is discussed and shown to be applicable for the EDGE architecture. This is followed by an exploration of how these hardware modifications affect performance of core composition under an idealistic scenario, where perfect value prediction is enabled. The benchmarks used in this chapter are the San-Diego Vision benchmark suite, the same as Chapter 6. Finally an evaluation of a value predictor [Pera 15], the block based differential VTAGE predictor (D-VTAGE) with the new fetching scheme is conducted.

To summarise, the contributions are:

- A presentation of a new fetching scheme, Round Robin Fetch that enables better utilisation of large core compositions.
- An analysis of how value prediction can improve performance of core composition on the SD-VBS benchmark suite [Venk 09].
- An implementation and evaluation of the block-based VTAGE value predictor for EDGE, demonstrating that performance can be improved by only predicting register reads.

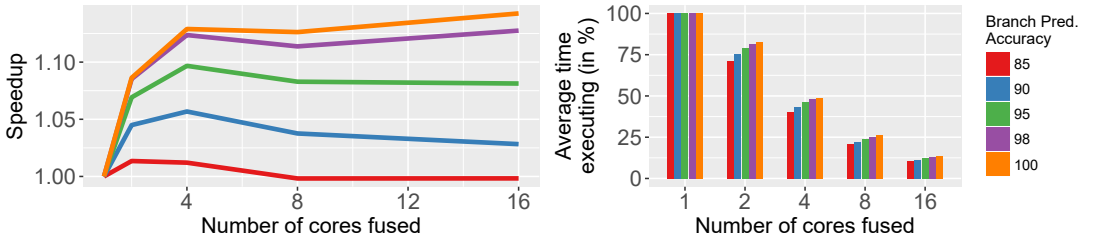


Figure 7.1: Left: Speedup obtained when executing the MSER benchmark on different compositions and branch prediction accuracies. Right: Percentage of time (in cycles) cores in a composition execute instructions compared to the overall execution time. Higher is better for both.

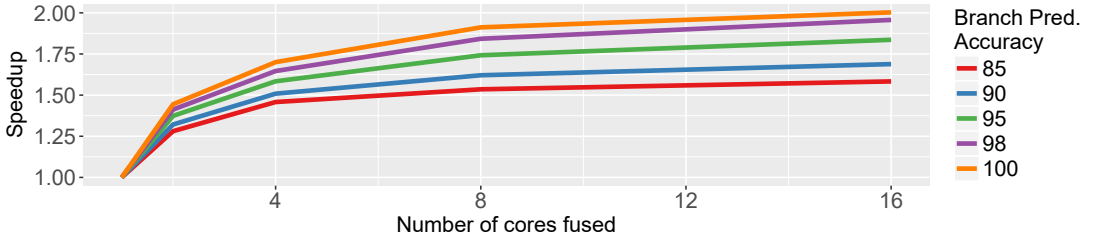


Figure 7.2: Speedup obtained when executing the MSER benchmark with different core composition with an oracle fetching scheme and perfect branch prediction. Higher is better.

7.2 Motivation

This section explores how core composition performance depends on branch prediction, fetching mechanism, and data dependencies between blocks and how modifications to certain mechanisms can improve performance.

7.2.1 Branch prediction

Chapter 6 section 6.2 highlighted the importance of branch prediction accuracy when fusing a high number of cores. To maximise utilisation, a core can have multiple blocks in its instruction window. In this thesis, the instruction window is segmented into four lanes, each of which can hold a block of up to 32 instructions. If the program executing mainly has small blocks, then if it is running on a 16 core composition, the branch prediction accuracy needs to be as high as 98% to ensure that the cores are fetching blocks on the correct execution path (see Chapter 6 section 6.2).

In Chapter 6, *MSER* had a low branch prediction accuracy of 85%, leading to a performance improvement of only 1% on a 2 core composition. To see how difference accuracies affect performance, a branch predictor that can predict at different accuracy

levels is used. The left hand side of Figure 7.1 shows the speedup obtained when executing the SD-VBS benchmark *MSER* on core compositions of size 2, 4, 8, 16 with different branch prediction accuracies on the cycle-accurate simulator. The speedup is obtained by comparing the performance to a single core. As the figure shows, increasing the accuracy to 100% leads to a performance increase of 1.15x on a 16 core composition.

7.2.2 Fetching mechanism

The reason performance does not improve much is due to the fact that *MSER* features small blocks. Currently, when cores are composed, they fetch blocks in a serial fashion, as defined in Chapter 2 Section 2.4.2.2. As cores only submit fetch requests to other cores in the composition if they are full, this means that if a core is able to commit a block before being full, then it will never submit a fetch request to another core. In this situation, cores in a composition may remain inactive during the execution of a program as they are not prompted to fetch blocks. Throughout the rest of this chapter, this fetching scheme is referred to as Serial Fetch (SF).

To understand how the fetching scheme can affect performance, the simulator records the number of cycles each core is actively executing code. The right hand side of figure 7.1 plots the average *active cycles* of cores in a 1, 2, 4, 8 and 16 core composition, compared to the total execution time in cycles using SF, with different branch prediction accuracies. The figure shows that increasing the size of a core composition when executing *MSER* will reduce the average time a core is executing a block. On a 16 core composition, each core is only actively executing a block 12.5% of the time. Cores are not being provisioned with blocks fast enough, thus, for a benchmark such as *MSER*, the current fetching scheme leads to inefficient use of large compositions.

To illustrate how modifying the fetching mechanism can improve performance of core composition, an oracle fetching mechanism (OF) is designed, in which cores can fetch in parallel and do not require any communication beyond receiving a prediction from another core. Figure 7.2 shows the speedup obtained by using the OF scheme on *MSER* with different branch prediction accuracies and a baseline of a single core. The figure shows that by modifying the fetching scheme a 16 core composition can potentially improve the performance of *MSER* by 2x, compared to the 1.15x obtained when using the SF scheme.

```
1  while( nvisited-- ) {
2       forest_pt [ sref(visited_pt,nvisited) ] .shortcut = nrindex ;
3     }
```

Listing 7.1: Example of loop found in MSER.

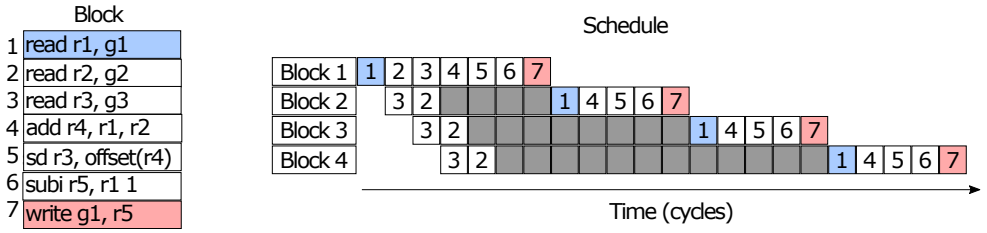


Figure 7.3: Example of how data-dependencies cause delays when executing four blocks in parallel. The numbers represent part of the loop body in Listing 7.1.

7.2.3 Data dependencies between blocks

In the EDGE architecture, physical registers are used for inter-block communication. For example, the code found in Listing 7.1 shows a loop found in the *MSER* benchmark when the value of the variable *nvisited*, which is used in both the header and loop body, will be passed from one block to another via a register read and write.

To illustrate the data-dependency problem, Figure 7.3 shows a simplified view of four blocks representing the body of the loop in Listing 7.1 being executed in parallel. Each block starts executing a cycle after its parent; the instructions highlighted in colour represent the register causing the data-dependency; blue represents the register being read, and red represents the register being written to. The grey slots represent cycles where the block cannot execute any instructions. Assuming each instruction takes a single cycle to execute, 22 cycles are needed to execute all four blocks in parallel, compared to 28 cycles if they were to be executed sequentially. If the data dependencies are not resolved quickly enough, then this causes blocks to execute in a serial fashion, which reduces any benefit from using the composition.

If cores do not have to wait on data dependencies, this can increase efficiency of core compositions, as they can execute their blocks independently. Figure 7.4 shows how an optimal configuration of the processor, one that can fetch blocks in parallel, has perfect branch prediction and can immediately resolve data dependencies can improve performance on *MSER*. The speedup is obtained by comparing the execution time of core compositions to a normal single core, without perfect branch prediction. The figure also shows the performance of a “normal” core composition configuration that has no perfect branch prediction, serialised block fetches and cannot resolve data depen-

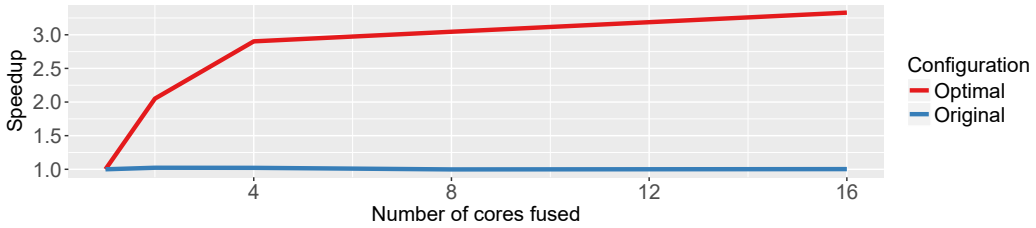


Figure 7.4: Speedup of executing *MSER* using the new fetching mechanism, with perfect value prediction and perfect branch prediction. Baseline is a single core with original branch prediction accuracy. Higher is better.

dencies immediately. As shown in the Figure 7.4, a 16 core composition can now get a speedup of up to 3x, compared to the 2x when using only perfect branch prediction and the OF scheme.

Serialised execution due to data dependencies is a common problem for superscalar processors [Pera 14]. One solution to the problem is adding a value predictor to the processor, which is able to predict the value of a register. This allows instructions to execute with speculative data, and thus increase ILP and reduce the impact of data-dependencies. Section 7.4.2 covers the implementation of a value predictor for an EDGE processor which is used throughout this chapter.

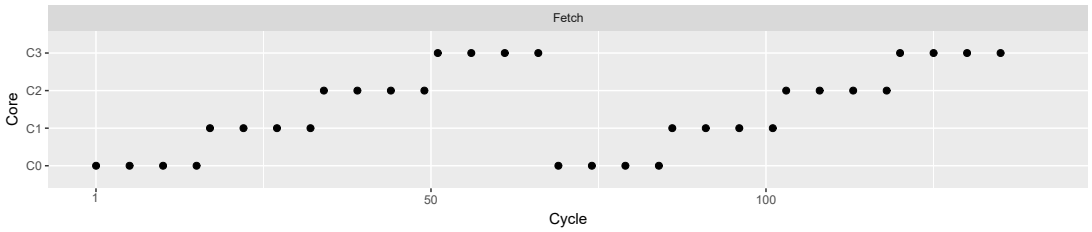
7.3 Round robin block fetching scheme

7.3.1 Current fetching scheme

The main issue with the SF scheme is that cores in a composition depend on each other in order to fetch blocks. For example, if Listing 7.2 is compiled without unrolling and executed on a core composition, each core has to fetch 4 iterations of the loop before sending the next fetch request to another core. To illustrate how fetching can be a bottleneck in this situation, instrumentation is added to the simulator to track when cores fetch blocks in order to be able to visualise how long it takes to fill up a core composition. Figure 7.3 plots when cores fetch blocks for a 4 core composition executing the code in Listing 7.2. Each point represents a block fetch, the X-axis represents the time (in cycles) whilst the Y axis represents which core has started fetching a block. The figure shows that there are 50 cycles between the first block fetched by Core 0 and the first block fetched by Core 3. Given that a block in Listing 7.2 takes only 10 cycles to execute, this means that Core 0 is inactive when Core 3 fetches. This is due to the fact that once Core 0 submits a fetch request to Core 1, it will have to wait for Core 3 to send it a fetch request.

```
1  for(int i =0 ; i < 100000; i++)
2    a[i] = c[i]*b[i];
```

Listing 7.2: Example of small loop.



Listing 7.3: Trace of when cores fetch blocks when executing Listing 7.2 on a 4 core composition. Y axis represents a core in the composition, X axis represents time.

7.3.2 Round-Robin-Fetching Scheme

This section demonstrates how the fetching mechanism can be modified to allow for cores to fetch blocks in parallel. It starts with a generalised version of the fetching algorithm for n cores in a composition. This is followed by a more in-detail example using a two core composition. Finally it compares the performance of the new fetching scheme with the current fetching scheme on a synthetic benchmark.

7.3.2.1 Generalised form

The advantage of core composition is that multiple cores can execute the same thread in parallel. Thus, the fact that the SF scheme prioritises filling a single core before using another core in the composition is counter-intuitive as it reduces the chance that all the cores are in use. The new fetching scheme has two design objectives: reduce the number of times cores depend on each other to fetch blocks and ensure that each core in the composition is always executing at least one block. Sequential blocks should not be found on the same core, instead they should all be on separate cores and fetched in a round robin fashion. This ensures a more equal distribution of work amongst all the cores in the composition.

If blocks are distributed equally amongst all cores using a round robin model, this may still require that each core must submit a fetch request to the next core. This means that changing the fetching scheme to a round robin scheme does not necessarily stop cores from depending on one another to fetch blocks. However, once a core has a block it can use branch prediction to predict the next block it must fetch, rather than waiting for another fetch request from a core. As this new fetching mechanism employs a

round robin scheme, the branch predictor must predict a block that is multiple steps into the future rather than the immediate branch. By allowing cores to fetch blocks in *strides*, instead of sequentially, the new fetching mechanism not only ensures cores have an equal amount of work (as long as the blocks are of equal size), but that they can fetch in parallel.

Fetching mechanism Algorithm 1 explains how the new fetching scheme, named Round Robin Fetch (RRF) works for n cores in a composition. In the general case, when a core composition is created, each core, aside from the core that initiated the composition, is empty. When a core fetches $block_i$, if the next core is empty, it makes two branch predictions, one for $block_{i+1}$ and one for $block_{i+n}$. The predictions do not have to be done on the same cycle, however previous work on multi-block ahead branch predictors demonstrated that two predictions per cycle is possible [Sezn 96]. The core submits a fetch request to the next core in the composition for $block_{i+1}$ and then uses the prediction $block_{i+n}$ to fetch its next block. If the next core is not empty, then the current core simply predicts $block_{i+n}$.

In case that a core cannot predict $block_{i+n}$, it simply submits a prediction for $block_{i+1}$ to the next core and continues to execute $block_i$. The core will then have to wait for another core in the composition to send it the prediction for $block_{i+n}$. Whilst this case may impact overall throughput, it is no different than the SF scheme as SF would also stop fetching after $block_{i+n-1}$ and wait for $block_{i+n}$'s PC to be resolved.

In the SF scheme, when a core is full, it submits a fetch request to another core in the composition by sending the address of a block to a buffer found on that core. To ensure cores are always fetching blocks in RRF, the buffer is used when a core has filled up its instruction window and can no longer fetch blocks. Instead of sending the fetch request to another core, a core saves the block address in its own buffer, and will handle that fetch request once it has committed a block. In this chapter, a core can only have one buffered PC at a time.

Committing mechanism Algorithm 2 shows the new steps added to committing blocks in RRF. When committing $block_i$, the core sets the next core in line to be the non-speculative core. This is due to the fact that the next core will always have $block_{i+1}$. If the next core in line does not yet have $block_{i+1}$ due to no prediction having been made, then the committing core will submit the resolved PC to the next core, rather than fetching it for itself. If the core has a PC in its buffer, it immediately starts

```
n = Number of cores in the composition
Composition[n] = Core Composition Array
branches[2] = Branch Predictions for current block
while Program is Executing do
    ▷ Do branch prediction for up to 2 blocks
    branches = prediction[i+1, i+n]
    ▷ If the predictor generated 2 predictions
    if size(branches) == 2 then
        ▷ If next core in composition is empty submit block i +
          1 to it
        if empty(Composition[currentPosition+1]) then
            submitToNextCore(branches[0])
            if myCore.full() == false then
                submitToMyself(branches[1])
            else
                buffer.push(branches[1])
            end
        else
            if myCore.full() == false then
                submitToMyself(branches[1])
            else
                buffer.push(branches[1])
            end
        end
    else if branches[0] is valid then
        ▷ If only block i + 1 prediction is valid
        if empty(Composition[currentPosition+1]) then
            submitToNextCore(branches[0])
        end
    end
end
```

Algorithm 1: Overview of fetching algorithm for *n* cores fused

```
n = Number of cores in the composition
Composition[n] = Core Composition Array
while Program is Executing do
    if block is committing then
        Composition[currentPosition+1] = Non Speculative;
    end
    if not IsBlockRunning(Composition[currentPosition+1], block+1) then
        submitToNextCore(block+1 PC)
    end
    if buffer.size() > 0 then
        fetch(buffer[0]) buffer.pop()
    end
    else
        Idle
    end
end
```

Algorithm 2: Partial overview of the commit stage for n cores fused

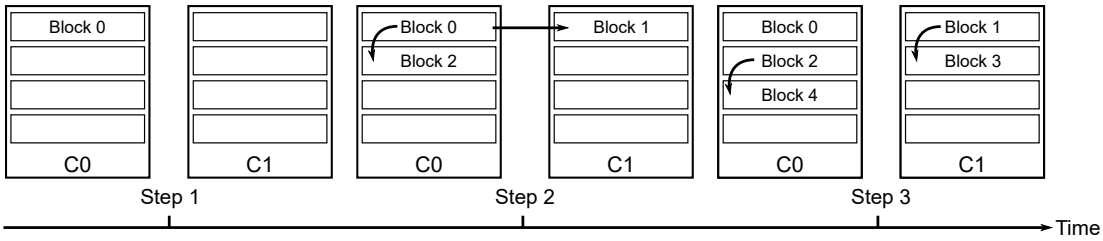


Figure 7.5: Example of RRF on a 2 core composition. Each core has 4 segments, the arrows represent the block generating the predictions.

fetching a new block. However, if it has no PC in its buffer, it must wait on another core to send it a request.

Two core example Figure 7.5 gives an overview of the first few cycles of using RRF with two cores fused. When $Core_0$ starts the composition and fetches the first block, $block_0$, if it is able to predict $block_2$ will submit a fetch request for $block_1$ on $Core_1$ whilst also attempting to fetch $block_2$ for itself. On the next cycle $Core_1$ receives the request for $block_1$ and starts fetching the block. Once $Core_1$ can make a branch prediction it will attempt to predict for $block_3$ instead of $block_2$; this is because $block_2$ was already predicted and fetched on $Core_0$.

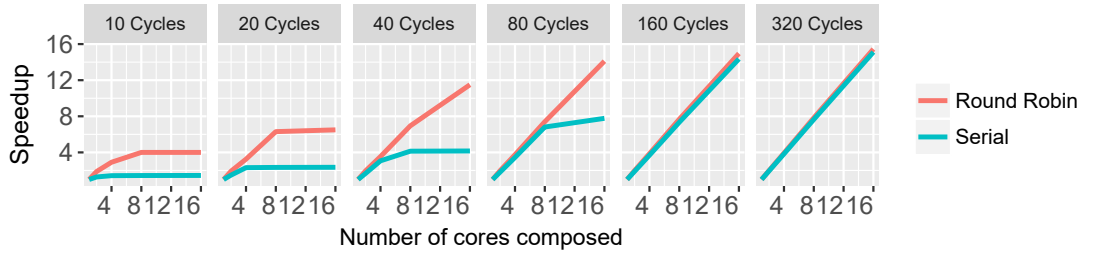


Figure 7.6: Speedup when executing the synthetic block with varying execution times (facets) with SF and RRF. Higher is better.

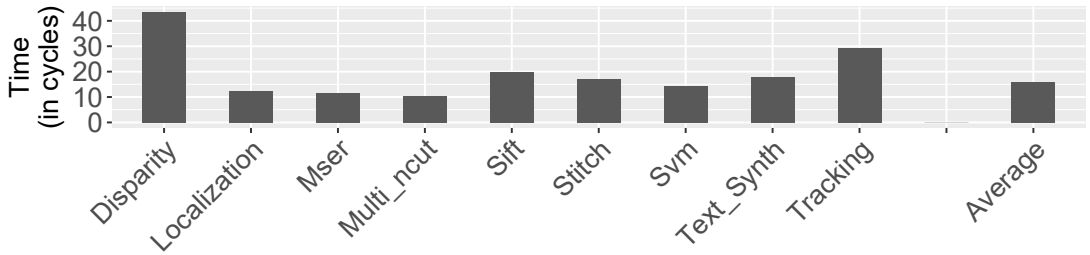


Figure 7.7: Average execution time (in cycles) of blocks for the SD-VBS benchmarks. Each benchmark is executed on one core with perfect branch prediction.

Limitation Whilst RRF allows for cores to fetch out of order in parallel, blocks must still be dispatched in order. Here, dispatch represents the moment an instruction can be sent to an ALU. This is to ensure that blocks do not execute data-dependent register reads out of order, when dependencies are not yet determined. To achieve this, each core maintains a flag that tells it whether or not its oldest - not yet dispatched block - can be dispatched. Since blocks are fetched in a round-robin fashion, whenever a core starts dispatching a block, it sets the flag to false, and informs the next core to start fetching. Only a single core has this flag set to true, thus ensuring blocks are dispatched in order. In this chapter, compositions are created out of cores that are physically close, and the following block is always on a core that is 1 network hop away (1 clock cycle). Assuming, for this architecture, that a hop takes a single cycle, a block can theoretically be dispatched every cycle.

7.3.3 Evaluating the round robin fetch scheme on a synthetic block

Before evaluating the performance of RRF on a set of benchmarks, it is important to measure the potential performance increase on a simpler case. This is due to the fact that other factors than the size of blocks, such as data-dependencies, cache misses or load-store queue violations can affect the performance of a composition. Therefore, evaluating the new fetching scheme on a synthetic benchmark can provide a ceiling for the maximum performance gains when using RRF.

For this section the synthetic block is only four instructions long using a custom instruction whose execution time is defined ahead of the simulation. The reason a four instruction block was chosen is due to the fact that it allows for four blocks to be fetched on each core, which is the worst-case scenario for the SF scheme. Whilst the SF scheme is susceptible to small blocks, the main issue is when these blocks execute quickly, as it means that the composition cannot be filled. On the other hand, if a block is small yet takes hundreds of cycles to execute, for example due to multiple cache misses, then the SF scheme would still perform fine, as filling cores would take less time than executing blocks. This is why the execution time of the block is variable, as it allows to cover different cases. For this experiment, six different execution times are explored: $Exec = \{10, 20, 40, 80, 160, 320\}$.

Figure 7.6 shows the speedup obtained when using core composition when executing the synthetic benchmark, with the SF or RRF scheme with a baseline of a single core. The facets represent the different execution times of the block whilst the colours represent the different fetching schemes. The results in the figure show that unless a block is at least 80 cycles long, it is difficult to efficiently use 16 core compositions using SF. To give a point of reference – with data gathered from the SD-VBS benchmarks from Chapter 6 – the average execution time (in cycles) of a block for each of the SD-VBS benchmarks is displayed in Figure 7.7. The figure shows that, on average each block in a benchmark only takes 15 cycles to execute, with *Disparity* having the longest blocks with an average of 40 cycles. With the new fetching scheme, 16 cores becomes useful when blocks are at least 40 cycles long, enabling a 3x speedup compared to the old fetching scheme.

7.4 Value Predictor

In the EDGE architecture registers are primarily used to pass values between blocks, which can cause data dependencies. In a core composition the register files of each core in the composition becomes distributed (Chapter 2 Section 2.4.2.2) to ensure that each core has the same view of the current execution state. An example of values passed between blocks is the loop induction variable of the loop in Listing 7.4.If this loop is executing on a 16 core composition, then one of the cores will have to issue all the reads to this register for all other cores. This of course will put stress on the NoC, and increase the latency of what is meant to be a fast instruction.

```
1  for(int i =0 ; i < 100000; i++)
2    a[i] = c[i]*b[i];
```

Listing 7.4: Example of small loop.

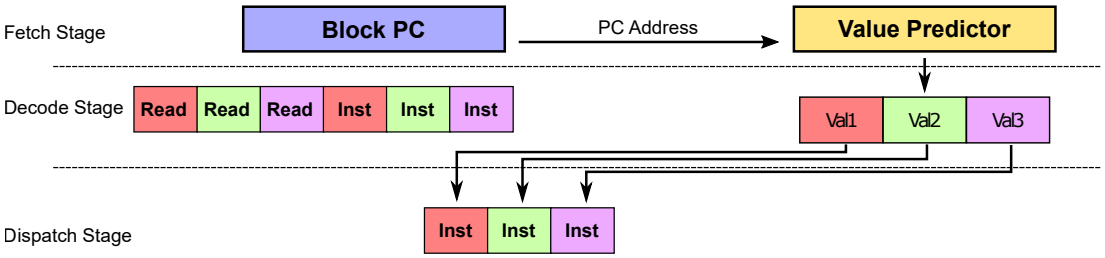


Figure 7.8: Overview of how a value predictor should work for EDGE. Prediction is made at the fetch stage, and predictions are used when register reads are dispatched.

To ensure that a younger block does not execute a read to a register that must be written to by an older block, the register-file keeps track of registers which will be written to by older blocks. If the younger block attempts to execute the read, its request is pushed back until the older block has executed its write and any instruction that depends on the read must wait until the write fires. Whilst the serialisation of register reads and writes between blocks ensures correct execution of speculative blocks, it effectively reduces the potential for instruction level parallelism (ILP). This is further exacerbated when fusing a large number of cores, as this increases the number of blocks that may have to wait on register reads and writes.

This chapter underlines two problems related to register reads on large compositions: the potential data dependencies caused by reads waiting for previous writes to execute, and the latency caused by having to send read requests via the NoC. The problem of trying to reduce register and memory dependencies to improve instruction level parallelism is not new, and is an issue found in more traditional out of order superscalar processors [Pera 14].

For example, in the loop of Listing 7.2, the sole data dependency is found in the loop induction variable. This variable is always incremented by 1, which means that given a block, the value can easily be predicted based on previous values of the variable. The other register values, such as the memory bases can also be predicted as they never change. If each core is able to predict the value of the register reads, then they can speculatively execute instructions that depend on these registers before the real value arrives. In these cases value prediction can be used to attempt to ensure that these blocks can run in parallel even if there are dependencies.

7.4.1 Design features of a value predictor

In this chapter, the only target for value prediction is register read instructions. This is due to the fact that they are prime suspects for data dependencies and unlike loads, currently cannot be fired speculatively. Ideally, a value predictor for EDGE would function as seen in Figure 7.8. When a block is fetched, a single request is made to the value predictor to fetch all predictions for the read instructions of the block. At dispatch time, the predicted values are used and forwarded to dependent instructions, whilst the read instructions are issued. This allows the depending instructions to execute whilst the reads are still being processed.

Prediction Latency In a traditional superscalar processor, one of the main challenges value predictors face is being able to sustain the potential number of prediction requests in a short time frame [Pera 15]. As value predictors are designed to improve ILP performance it is important to be able to issue a predicted value quickly. If multiple prediction requests are made each cycle, this requires expensive hardware such as a re-order buffer to hold all predictions [Pera 15].

To tackle the challenge of issuing predictions quickly, research has focused on grouping predictions into blocks [Pera 15]. Instead of issuing a request per instruction a single prediction, the predictor receives a single request for a set of instructions in a basic block. Entries are accessed by using the PC of the first instruction of the fetch block. By grouping multiple predictions into a single entry, it drastically reduces the number of requests to the value predictor, reducing the prediction latency for a large number of instructions. As EDGE organises instructions as blocks, a block-based predictor would reduce the number of prediction requests per cycle, making it an attractive feature.

Prediction generation Another important feature when selecting a value predictor is how it generates a predicted value. Currently, there exist two methods: *context* value prediction [Pera 14] and *computational* value prediction [Pera 15, Gabb 98, Goem 01]. More details on how these two predictors differ can be found in Chapter 2 Section 2.5. To summarise, *context* predictors simply fetch a value from a table to generate a prediction, whilst *computational* calculate the predicted variable.

Whilst *context* predictors are simpler to design, as there are no extra functions required to generate the predictions aside from fetching a value from a table, they are often considered less efficient when used in loops [Pera 15]. For example, the

memory address of an item in an array from Listing 7.4 is always incremented using a fixed stride (the loop induction increment). In order for a *context* value predictor to correctly predict the address, it must already be stored in its table. Multiple values for a single instruction must be stored in the table in order to predict the different addresses throughout the execution of the loop. Having a large number of values stored in the predictor for a single instruction is inefficient, unless the predictor is very large.

On the other hand, a *computational* predictor can capture how the memory address changes each iteration by determining the *stride* at which it is modified. This means that a *computational* predictor will only have to store 2 values: one for the stride and another for the last committed value. Not only does this reduce the number of entries a single instruction occupies in the predictor, but it allows a *computational* predictor to generate predictions faster than the *context* predictor since the next memory address will be equal to $lastCommittedValue + stride$.

Core composition is often most effective when executing over loops, as seen throughout chapter 6. As variables such as loop inductors are passed between blocks via register reads and writes, they are a prime candidate for value prediction. Since these variables are often modified using the same stride throughout the loop and the loops may occur multiple times during the execution of the program it is important that the predictor keeps the least amount of information for each value. This is to ensure that multiple values can co-exist in the predictor, increasing the overall coverage. *Computational* predictors are therefore more adequate for this scenario, as they require fewer entries to predict a single value and are able to generate predictions faster.

Summary A value predictor for EDGE must be able to provide predictions in groups, as EDGE organises its instructions in blocks as this reduces the number of prediction requests per block. As core composition is mostly used to improve the performance of loops, a *computational* predictor is more adequate than a context based one. Perais *et al.* propose such a predictor: a block based differential Value TAGE predictor [Pera 15]. The next section covers briefly how this predictor works, however more details can be found in Chapter 2 Section 2.5.

7.4.2 Block-based D-VTAGE predictor

In this chapter, a *block* based differential value TAGE (D-VTAGE) predictor is implemented, based on the work of Perais *et al.* [Pera 15]. Full details on how such a value

predictor works can be found in Chapter 2 Section 2.5.1. To summarise, D-VTAGE is a *computational* based value predictor: a prediction is composed of the last seen value for the instruction (found in a Last Value Table (LVT)), and a stride which represents the delta between the last two values for the instruction. When a prediction is made, the last seen value and stride are added together to make the predicted value. Unlike a traditional value predictor that issues 1 prediction per instruction, this predictor issues multiple predictions at a time (*blocks* of predictions). Predictions are validated during the commit phase of a block [Pera 14]. This is achieved by comparing the predicted values with the real values read from the registers. In case of a misprediction, a pipeline squash is issued for all blocks younger than the mispredicting block. The mispredicting block is then re-executed, this time without value prediction.

The predictor also handles the fact that multiple blocks may be in flight, and thus the value found in the LVT may not be up to date. This is done via a speculative window that has its own LVT that is speculatively updated when new predictions are made. This allows the predictor to be able to handle situations where multiple iterations of a loop are in flight. Since EDGE blocks are single entry, there is no need for a complicated update mechanism for the Speculative Window as defined in [Pera 15]. When a block is flushed, it is also removed from the speculative window.

7.5 Experimental Setup

7.5.1 Benchmarks

To evaluate how the hardware modifications improve the performance of core composition; the same benchmarks used in Chapter 6 are used here. These benchmarks are all from the San-Diego Vision Benchmark Suite (SD-VBS) [Venk 09], which is composed of a set of vision and image analysis applications, and are described in detail in Chapter 4 Section 4.2.2. The previous chapter showed that even with code optimisations, core compositions do not perform optimally when executing the benchmarks. Some of the programs, such as *MSER* or *Multi_NCut* features an average block size of under 10 instructions, making it difficult to use core composition efficiently. They are therefore a perfect candidate to explore how the hardware modifications can improve the performance of core composition.

	Disparity	Localization	MSER	Multi_NCut	Sift
Input	VGA	VGA	CIF	SIM_FAST	CIF
	Stitch	SVM	Text. Synth	Tracking	
Input	CIF	CIF	FULLHD	VGA	

Table 7.1: Datasets used for each of the benchmarks.

7.5.2 Evaluation

The previous chapter showed that the SD-VBS benchmarks feature repeating phases of IPC. These benchmarks are structured as pipelines with distinct passes that are often repeated. This means that performance improvements can be analysed without having to fully execute the program.

As this chapter is only concerned with demonstrating that the new hardware modifications outperform the current implementation, the benchmarks are executed long enough to capture all the phases. The phase data gathered from Chapter 6 is used to determine hot-spots. The benchmarks are then instrumented so that the main phases are captured, and executed for at least 100 million instructions. Of that 100 million instructions, 10 million instructions are for warming up the caches and predictor whilst the rest are used to record performance. Finally, the same data-sets from Chapter 6 are used, to maintain a consistency amongst the thesis; they can be found in Table 7.1.

7.5.3 Value Predictor

Value predictors that generate predictions both quickly and accurately are still actively being researched [Pera 14, Pera 15, Shei 17]. In order to motivate the use of value prediction for core composition, it is first important to abstract away current implementation details of state of the art predictors. By considering a 100% accurate prediction rate and immediate value prediction, this helps to determine how much value prediction can help improve performance. Once the maximum speedup is determined, using a current state-of-the art implementation can help understand how far current value predictors are from the best performance.

This chapter explores two value predictors: a perfect value predictor that can predict any value in a single cycle, and a block based D-VTAGE [Pera 15] value predictor. To have a better picture of how state-of-the art affects performance, different parameters of the predictor are modified. This is discussed in further details in Section 7.7.4.

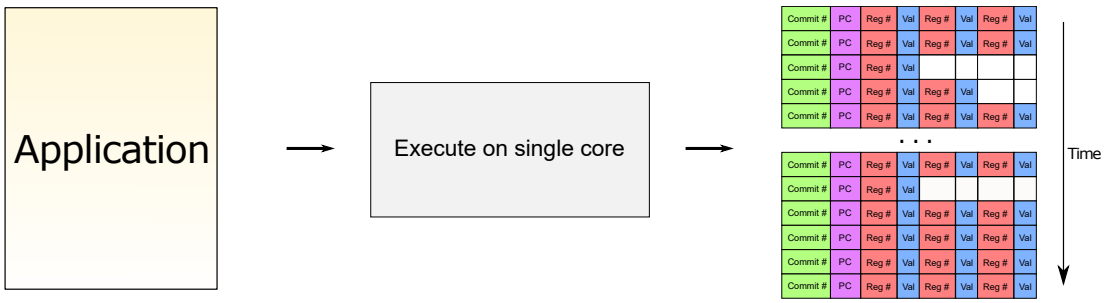


Figure 7.9: Overview of information gathering for generating traces which are used for the perfect branch and value predictors.

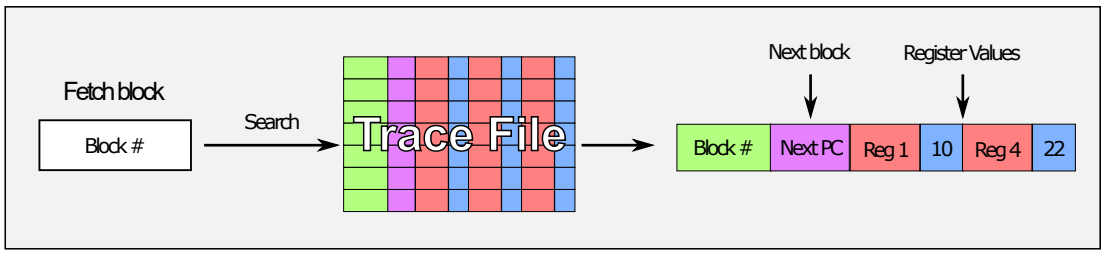


Figure 7.10: Overview of how the trace data generated for value prediction is used during execution of a block.

7.5.4 Implementing perfect value and branch predictor

This chapter is concerned with how changing the hardware will improve the performance of core composition. In order to motivate new research in branch prediction and value predictors and their use for core composition, it is essential to know what the maximum performance improvements are when using these techniques. Thus, a perfect value predictor and branch predictor are considered for the analysis.

These predictors use execution traces of each application to make their predictions. Figure 7.9 shows how these traces are generated. The trace file contains an entry for every committed block, which is comprised of the Program Counter (PC) for the next block, a list of registers that were read and their corresponding values.

When the perfect predictors are activated, the simulator reads in the trace file. Figure 7.10 shows how the trace data is used: a newly fetched block is paired with its correspondent trace data. Instead of using the branch predictor, the next block's PC is directly taken from the trace entry, and whenever the block can issue a register read, the value is fetched from the trace entry instead of making a request to the register file.

7.6 Analysis using perfect value prediction

This section explores how perfect branch and value predictors, paired with the new fetching scheme (RRF), improves the performance of core composition. To understand how each component contributes to the performance improvements, different configurations were used, they are as follows:

- Serial fetching scheme with no value prediction (***SFNoVP***).
- Serial fetching scheme with perfect value prediction (***SFVP***).
- Round robin fetching scheme with no value prediction (***RRFNoVP***).
- Round robin fetching scheme with value prediction (***RRFVP***).

All configurations use perfect branch prediction to ensure that core composition is always on the correct execution path. All benchmarks are executed with 16 cores composed as this is the maximum number of cores that can be fused. No dynamic adaptation is done as Chapter 6 showed that the primary advantage of dynamic core composition is energy savings, whereas this chapter focuses on speedup.

7.6.1 Analysing the performance of the different configurations

Figure 7.11 shows the speedup obtained on the SD-VBS benchmarks using the different configurations. The baseline for this section is 16 cores composed with serial fetch (SF) no value prediction (***SFNoVP***) and perfect branch prediction. This baseline is chosen as this chapter is focused on improving the performance of core composition.

First, it is clear that using RRF with value prediction (***RRFVP***) always results in the best speedup compared to the baseline. For *Multi_NCut*, performance is improved by 3x when using ***RRFVP***. This is a significant speedup, as Chapter 6 showed that *Multi_NCut* is a difficult benchmark for core composition (1.3x speedup in Chapter 6). On average, ***RRFVP*** outperforms the baseline by a factor of 1.88x.

7.6.1.1 Performance without value prediction

The results in Figure 7.11 show that without value prediction the performance improvements brought by RRF on its own are low, or in fact detrimental to performance. For example, *Multi_NCut* only has a 1.10x speedup when using the ***RRFNoVP*** configuration, compared to the 3x of ***RRFVP***. This is due to the fact that whilst more blocks are now spread across cores, the register dependencies between blocks limit the performance of the composition. In fact, the more even distribution is the reason why some benchmarks perform worse: *Disparity*, *Texture_Synthesis* and *Tracking* see

Figure 7.11: Comparing the performance of serial fetch to round robin fetch, with and without perfect value prediction. Higher is better.

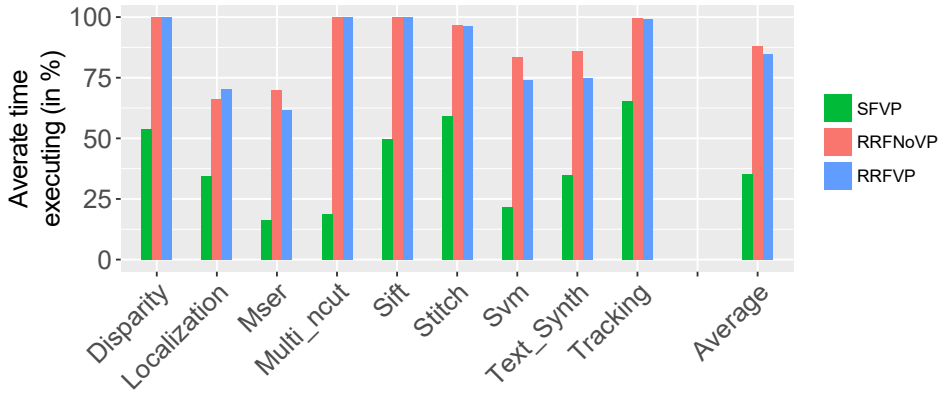


Figure 7.12: Average time each core is executing blocks (in %) for each benchmark, using the different configurations. Higher is better.

a slight performance decrease. The even distribution of blocks amongst cores increases the stress on the network on chip (NoC) as more cores will make accesses in parallel. Without value prediction, RRF suffers due to NoC stress and data dependencies.

7.6.1.2 Performance with value prediction

The performance improvements brought by RRF are more apparent when taking value prediction into account. ***RRFVP*** has a 54% performance increase compared to ***SFVP*** (1.88x vs 1.22x). The difference in performance comes from the fact that with value prediction, blocks can potentially execute faster, and thus a faster fetching scheme is required to keep up. Since the SF scheme is slower, it is less likely going to benefit from parallel execution. Even so, ***SFVP*** results in a 1.5x speedup for *Localization* which shows that value prediction is valuable for composition regardless of the scheme.

Active Cycles To better highlight how the SF scheme hinders performance even with value prediction, the percentage of time a core in a composition is actively executing code, for each benchmark, is shown in Figure 7.12. For each configuration, the number of cycles each core in a composition has instructions to execute is averaged out and then compared to the total execution time of the application. When the average active time of a core is close to the total execution time, this means that the composition was efficiently used, as each core had a block to run throughout the program execution.

Figure 7.12 shows that ***SFVP*** often has low active times when using a composition of 16 cores. This is due to the fact that the SF scheme is slow, and thus, some cores are inactive, waiting to receive a fetch request from another core. The lower the percentage is, the less likely there are going to be multiple blocks on different cores in flight which in turn means the composition is less efficient and value prediction is less useful. Since value prediction is aimed at increasing instruction level parallelism (ILP) [Pera 15] it is important that cores may fetch blocks quickly in a composition.

With the RRF scheme, the percentage of active time is increased on average by a factor of 2.28x, and is on average 85%. This means that during most of the execution of an application, all cores are executing code, and thus greatly increases the chance of improving performance via core composition. It is interesting to see that ***RRFVP*** has a lower average time than ***RRFNoVP*** for some benchmarks such as *MSER* and *Texture_Synthesis*. Also, whilst RRF aims to evenly distribute blocks amongst the cores, the average core utilisation for *Localization* with the configurations ***RRFNoVP*** and ***RRFVP*** is of 62%. This reduced average time is due to flushes caused by Load-Store-Queue (LSQ) violations, which causes cores to flush their instruction windows, and thus increases the number of times that cores will not be executing code. Figure 7.13 shows the number of blocks that cause an LSQ violation, normalised by the number of fetched blocks for each of the benchmarks. Even though the percentage of violations is small, it still has an impact on how efficient the composition is, due to the fact that compositions rely on heavy speculation to obtain any performance improvements.

7.6.1.3 Summary

This section shows that a perfect branch predictor, paired with perfect value prediction and the round robin fetching scheme can outperform the current configuration by a factor of up to 3x and on average a speedup of 1.87x.This section also showed that without value prediction RRF only obtains a 1.09x speedup compared to the serial fetching scheme. This is due to the fact that the benchmarks all display a certain

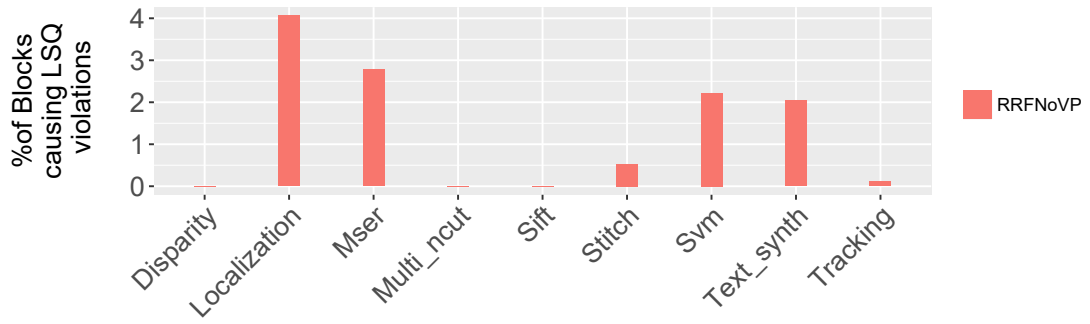


Figure 7.13: Number of blocks that cause LSQ violations, normalised by the number of fetched blocks for each of the benchmarks.

amount of data-dependencies between blocks and that spreading blocks across cores more evenly can put pressure on the NoC; thus reduce the performance improvements of fetching blocks quicker.

7.7 Analysis using the block based D-VTAGE predictor

This section evaluates a real implementation of a value predictor, using the block based D-VTAGE predictor. Before conducting the performance analysis, it is important to understand how the predictor must be configured. This section therefore starts with a block analysis of the SD-VBS benchmarks.

7.7.1 Block analysis

In a block based D-VTAGE predictor a single prediction request fetches multiple values. Due to the fact that the block size is fixed ahead of time [Pera 15], determining the correct size is important. Having a block with many values means that the D-VTAGE predictor can predict a higher number of register reads for a single EDGE block, however this comes at the expense of having fewer blocks in the predictor tables. On the other hand, a small block size allows to have multiple blocks stored at a time but means that not all instructions can have their values predicted in the EDGE block. This could be potentially remediated by allowing a single EDGE block to occupy multiple predictor blocks, however this technique is not explored in this thesis.

To determine the size, all the EDGE blocks of the SD-VBS benchmarks are analysed to determine the average unique register read and write count per block throughout the execution of the benchmarks. The writes are tracked as it provides information on the potential number of register dependencies between blocks found in each of the

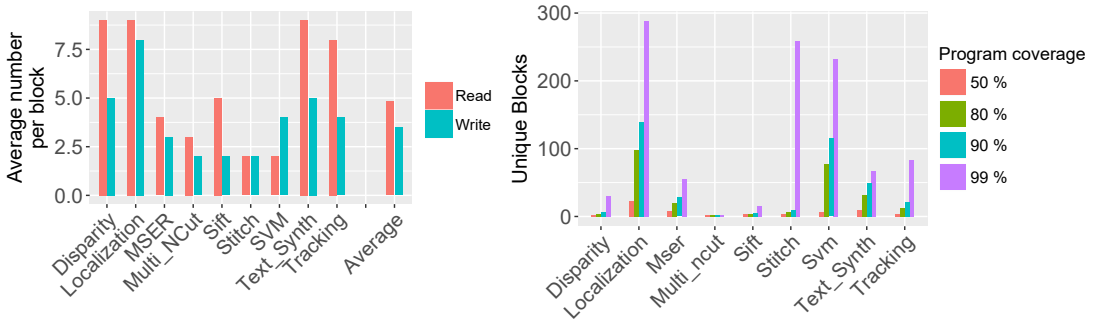


Figure 7.14: Average number of register reads and writes per EDGE block and the number of unique blocks comprising different percentages of the total execution (in blocks) of each of the benchmarks.

applications. The left hand side of figure 7.14 shows the average number of register read and writes for each of the benchmarks. Whilst *Disparity Localization*, *Texture_Synthesis* and *Tracking* have higher register read counts than the average, most of them only have 5 writes per block. This means that having a D-VTAGE block size of 5 could capture all dependencies; however this would require detection of which reads are data-dependent, which is beyond the scope of this thesis. As Section 7.6 showed these benchmarks benefit from value prediction, to ensure that most blocks have all their register reads captured, a block size of at least 8 is required.

Block variation analysis One method of understanding how the block based D-VTAGE predictor will perform is to study the number of unique blocks found in each of the benchmarks. Benchmarks that feature a smaller number of unique blocks can potentially benefit more from value prediction as the predictor cannot hold many blocks at a time. Reporting all unique blocks executed is not a proper evaluation of the variety of blocks found in the benchmark, as some will most certainly be executed more times than others. To account for this, blocks only count towards the unique block count as long as they represent a certain percentage of the total number of blocks executed.

The right hand side of figure 7.14 shows the number of unique blocks found in each benchmark that account for 50%,80%,90% and 99% of the total number of executed blocks. As can be seen, applications *Disparity*, *Multi_NCut* and *Sift Stitch* and *Tracking* execute fewer than 50 unique blocks during 90% of its total execution. This is promising as it means that there is a high chance that the predictor requires fewer entries to capture all possible blocks in the application. On the other hand, *Localization*, and *SVM* execute over 100 blocks throughout 90% of their execution, twice as many

Parameter	Values
# Base Entry	1536
# Tagged	6×1536
# Block in Spec Window	4
# of values per entry	8, 16
Confidence Value	4 or 7 with FPC

Table 7.2: D-VTAGE table configuration and configurable parameters.

as the previously mentioned benchmarks. For these applications, it might be harder to predict values as new blocks may overwrite entries in the predictor.

7.7.2 Setup

This section demonstrates how the block based D-VTAGE value predictor improves the performance of core composition using the round-robin fetch scheme (RRF). Serial fetch is not explored since the previous section showed that when using value prediction, RRF always provides the best results. The results reported in this section represent the culmination of hardware modifications for core composition discussed in this chapter. All benchmarks are executed using a 16 core composition.

Predictor size The D-VTAGE size configuration can be found in Table 7.2. The Last-Value Table shares the same number of values as the Base Entry and uses a 5-bit tag. Each tagged entry has a tag of varying size (first table is 12 bits, second 13 bits so on and so forth). The total number of values for the D-VTAGE predictor is taken from Perais *et al.*'s work [Pera 15]. The adopted size for this chapter is the medium predictor as it provides a good compromise between performance and total size of the predictor. The blocks in the speculative window represents the total number of in-flight blocks that can be handled by the speculative window, which is 4.

On area and power/energy consumption Since value prediction is still an experimental piece of hardware, this chapter focuses on demonstrating that they are useful in the context of core composition. This chapter is therefore a limit study of the performance improvements obtained via value predictors in large core compositions. This study motivates the necessity of value predictors, to push further research into how to efficiently implement them in the hardware. Therefore, no energy/power consumption or area information is provided here.

Parameter tuning Two features of the predictor can be modified: the number of values per entry and at what confidence a prediction is used. Modifying the required confidence explores the trade-off between high coverage and low misprediction. The original D-VTAGE uses Forward Probabilistic Counters (FPC) [N 06] to increment the confidence, and only used predictions once the counter was set to 7. This confidence rate ensured that the predictor had an accuracy of over 99%, but at the expense of a low coverage, 20% on average [Pera 15]. The confidence of 4 is chosen as it allows for very fast deployment of predictions, whilst still ensuring that the values have been trained at least a few times. The FPC vector used in this Chapter is the same as the one in Perais *et al.*'s work $\{1, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32}\}$.

Block size Whilst most applications only have 5 reads per block, *Disparity* and *Localization* have at least 8 reads and they both benefit from value prediction as seen in Section 7.6. Since there is no method of determining which reads may have data-dependencies, it is important to have a block size that captures the largest register read average for the set of benchmarks which is 8. This section explores block sizes of 8 and 16 to ensure that all register reads are potentially covered. The block size of 16 is explored to see how capturing a higher number than the average number of reads affects the performance of the predictor.

Prediction Delay In the original D-VTAGE proposal, they suggest that the predictor can generate x values per cycle where x is the issue width of the core. This means that 8 values can be generated per cycle, as this is the issue width of a core (see Chapter 4 Section 4.1). Since a request to the value predictor can be made at the same time as the start of a block fetch as seen in Figure 7.8, this gives enough time to fetch all the values before a block is fully dispatched as most blocks have at least 8 instructions.

7.7.3 Results

Performance Figure 7.15 reports the speedup obtained with the different configurations of the D-VTAGE predictor with RRF. The baseline is a 16 core composition that uses the serial fetching (SF) scheme without any value prediction and using perfect branch prediction. To better understand the performance of the predictors, each configuration is compared to the perfect predictor, which can predict every register value at any instance. Throughout this section, the configurations of D-VTAGE are labelled: **D-VTAGE_*Confidence*_*BlockSize***.

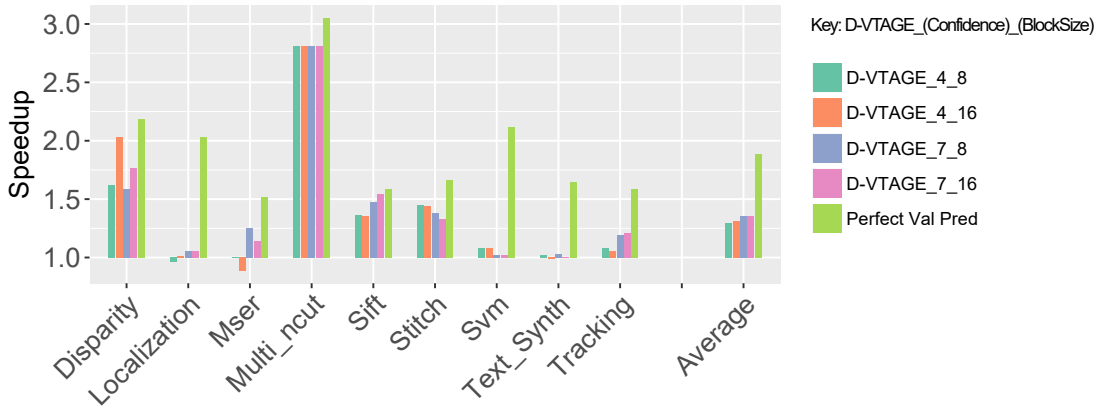


Figure 7.15: Speedup obtained using a D-VTAGE value predictor and RRF with 16 cores composed. Baseline is 16 cores composed with SF and without value prediction. Both use perfect branch prediction. Higher is better.

On average, using a D-VTAGE value predictor with the RRF scheme results in an average speedup of 1.32x. When comparing the performance between the different configurations, the main observation is that using a higher confidence results in better performance, however slight, 1.35x for D-VTAGE_7_16 compared to 1.31x for D-VTAGE_4_16. However, for the benchmark *Disparity* using a higher confidence leads to less of a performance increase 1.75x speedup compared to 2.0x for a confidence counter of 4. This is due to the fact that *Disparity* is composed of loops with highly predictable values, and thus deploying predictions earlier results in better performance improvements without risking a higher misprediction rate. On the other hand, *Localization* and *MSER* perform worse with a lower confidence counter, and result in a slowdown. Benchmarks *Sift* and *Tracking* perform better with a higher confidence, but the results of using a 4 bit counter never negatively impact performance.

The 16 values per block does not lead to better performance than 8 values but stays on par with it. The only noticeable difference is with *MSER* where the smaller block performs better. A lower number of values per block allows the predictor to capture a higher variety of executed blocks. Figure 7.14 shows that *MSER* has at least 50 different blocks throughout 99% of its execution, and on average, only 3 register reads per block. Therefore, for *MSER*, a predictor with fewer values per entry can capture more of the variation. On the other hand, for *Disparity*, 16 values per entry is required to get the best performance. This once again corroborates with the information from Figure 7.14. *Disparity* has a higher average of reads per block, but features a very low number of unique blocks. Therefore having a larger entry in the table captures more values, but does not suffer from having entries being replaced by new blocks.

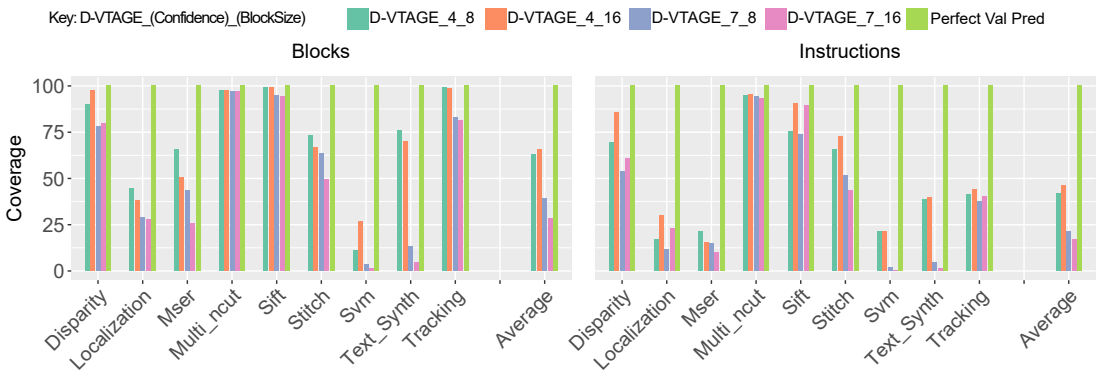


Figure 7.16: Prediction coverage at a block and instruction level. Higher is better

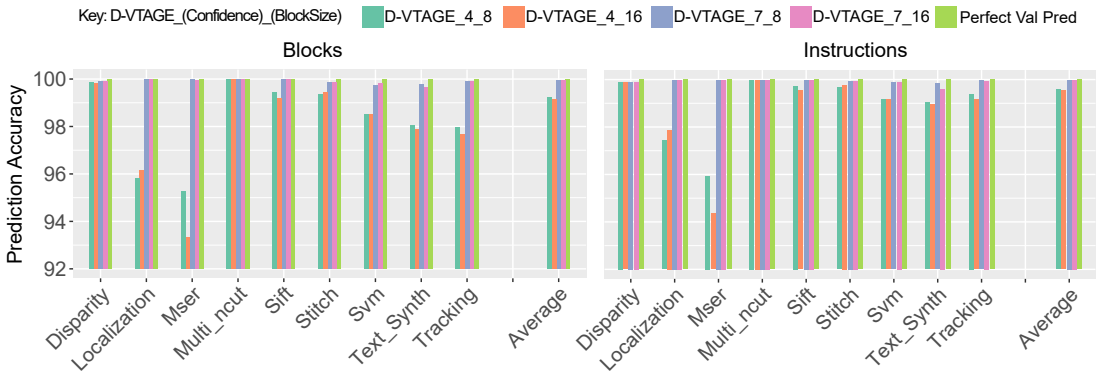


Figure 7.17: Accuracy of the different D-VTAGE predictors at a block and instruction level. Higher is better.

Comparing the performance of the different configurations to the perfect value predictor, it is apparent that most benchmarks do not achieve their maximum potential as the perfect predictor results in an average speedup of 1.88x compared to 1.32x of D-VTAGE. However, benchmarks *Disparity* and *Multi_NCut* show how value prediction with RRF is a promising lead for improving the performance of core composition.

Coverage and Accuracy To better understand the performance of the different D-VTAGE configurations, the predictors' coverage and misprediction rates are recorded. The coverage is measured by comparing the number of correct register read values to the total number of executed register reads. Since EDGE is a block based architecture, it is important to study the coverage and accuracy on both a per-instruction level and per-block level. This is due to the fact that a single read misprediction leads to flushing the block and all younger blocks in the chain. Also a single accurate register prediction may not necessarily improve performance as other un-predicted registers in the block may be data-dependent with other blocks, reducing ILP.

In this chapter, only blocks that are committed count towards the predicted blocks as any flush will artificially inflate the coverage count. Figure 7.16 displays the number of blocks which have at least one prediction, relative to the total number of committed blocks and the number of predicted register reads relative to the total executed register reads. The figure shows similar patterns for both coverages: the predictors with lower confidence counters have a higher coverage, 65% compared to 31% for the high confidence counter. This is normal: lower confidence allows predictions to be used earlier, which will increase coverage.

Whilst the block coverage is high, the coverage for registers reveals that not all registers are being predicted. Once again, lower confidence equates to higher coverage, however this time it's 30% for a confidence of 4 and under 25% for a confidence of 7 with FPC. The register coverage for a confidence of 7 is in line with the coverage reported in Perais. *et al.*'s work [Pera 15, Pera 14] if not slightly higher. The coverage may be slightly higher due to the fact that the values being predicted are often either loop increments or memory increments which can easily be predicted.

The register coverage helps explain why the high block coverage does not lead to better performance: whilst most blocks may have a valid prediction, some of the register reads cannot be predicted. This may be due to data being carried over which is unpredictable; for instance a loop which sums all values of an array into a single integer will pass that integer via a register. This register will cause a data dependency between loop iterations and will be difficult to predict.

Localization, *MSER*, *SVM* and *Texture_Synthesis* often have much lower coverage than the rest of the programs. Recalling Figure 7.14 which shows the number of unique blocks executed throughout the programs, these benchmarks had a higher count of unique blocks and thus a higher chance of encountering a new block. As blocks require multiple executions to train the predictor (if the values in the register are indeed predictable), than the higher diversity of blocks makes it harder. Thus, these benchmarks will naturally have a harder time to benefit from value prediction.

Figure 7.16 shows that a lower confidence allows for higher coverage, yet the speedups seen in Figure 7.15 indicate that a higher confidence leads to better performance. To confirm this, Figure 7.17 presents the prediction accuracy rate for each benchmark at a per-block and per-instruction base respectively. The D-VTAGE predictor maintains a 99% accuracy, whether at the block level or instruction level. However, for *Localization*, *MSER*, *SVM*, *Texture_Synthesis* and *Tracking*, the block level accuracy for a confidence of 4 is under 98%, and even down to 93% for *MSER*. The effect

Disparity	Localization	MSER	Multi_NCut	Sift
98%	95%	85%	100%	99%
Stitch	SVM	Text. Synth	Tracking	
95%	93%	98%	98 %	

Table 7.3: Branch prediction accuracy in percentage for each of the benchmarks.

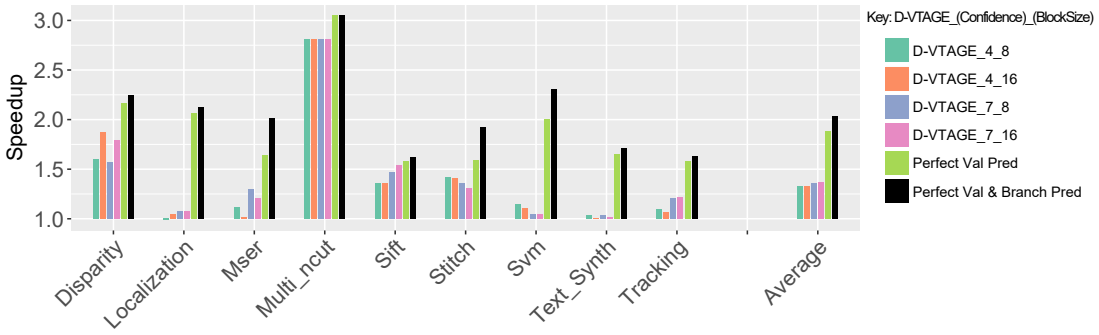


Figure 7.18: Speedup obtained using a D-VTAGE value predictor and RRF with 16 cores composed. Baseline is 16 cores composed with SF and without value prediction. Both use a non-perfect branch predictor. Higher is better.

of a value misprediction is similar to a branch misprediction: it causes a flush of all blocks younger than the block with the incorrect misprediction. Just like branch prediction, large core compositions are very sensitive to mispredictions, thus an accuracy rate of 93% will impact performance if the blocks are small. This explains why the low confidence counter sometimes performs worse than the higher confidence as it mispredicts more often.

7.7.4 Performance with non-perfect branch prediction

Finally, this section studies how non-perfect branch prediction affects the performance of the applications with RRF and value prediction. To conduct this analysis, the perfect branch predictor was modified to randomly mispredict given a certain accuracy requirement. For each of the benchmarks, the branch prediction accuracies recorded in Chapter 6 are used. The modified branch predictor is used to ensure that the same accuracies are maintained, even with the potential influence of a value predictor. These accuracies can be found in Table 7.3.

Figure 7.18 reports the speedup obtained with the different configurations using a non-perfect branch predictor. In this scenario, the baseline is a 16 core composition with SF and no value prediction, using the branch prediction accuracies defined in Ta-

ble 7.3. The figure shows an extra configuration that shows the results of the perfect value prediction with perfect branch prediction, to give perspective of how branch prediction affects performance. As the figure shows, the new configuration still performs well, due to the fact that most benchmarks had either high enough branch prediction accuracies, or blocks large enough to sustain 16 composed cores. This confirms that even without perfect branch and value prediction, performance can be improved by up to 2.8x, with an average of 1.30x. This is to be expected as branch prediction has a long history of research whilst value prediction has only recently seen a resurgence in interest. The results encourage more development in value prediction, as the perfect value predictors show there is still a lot of performance to be gained.

7.8 Conclusion

This chapter has investigated how new hardware can be used to improve core composition performance. Due to the fact that blocks are fetched in a serial fashion, when a large number of cores is composed this can severely reduce the amount of time that all cores are actively executing blocks. Therefore, finding a way of allowing cores to fetch in parallel can help increase the effectiveness of a composition. However, another issue arises: inter-block communication via register reads and writes can cause potential data-dependencies which serialises block execution. As register files are distributed when cores are composed, cores may have to send read requests to cores that are physically far away, increasing the latency of read instructions. Finding a way to predict data can potentially alleviate data dependencies and reduce the effect of high latency reads, improving the performance of core composition.

This chapter proposed a round-robin fetching mechanism, where cores do not fetch sequential blocks, but rather fetch blocks in strides. This enables cores to fetch independently from one another, without having to submit fetch requests to other cores. Using such a scheme can potentially increase the performance of large core compositions on small blocks by a factor of 2x to 3x. Then the idea of using a block based value predictor was covered. This value predictor was initially designed by Perais. *et al.* [Pera 14, Pera 15]. The design choices behind the value predictor match some of the architectural features of EDGE, mainly to do predictions at the granularity of a block.

To understand how these two additions can impact performance, the same set of benchmarks used in Chapter 6 are explored here. First, the performance of a perfect

value predictor with the round robin fetching scheme is explored and shows that it can improve the performance of core composition by up to 3x, with an average of 1.88x. This was followed by an analysis of the performance of using different configurations of the D-VTAGE value predictor with and without perfect branch prediction. Overall, using state-of-the art value prediction with round-robin fetching scheme leads to a performance improvement of up to 2.7x with an average of 1.3x even without perfect branch prediction compared to serial fetch without value prediction. This motivates further research in value prediction for core composition as it is an effective way of improving performance. To summarise, the contributions of the chapter are:

- The proposal of a round robin fetching scheme where cores fetch in parallel, out of order, and dispatch in order, allowing for an average improvement of 50% over compared to serial fetching scheme with value prediction.
- Demonstration that by only predicting register read instructions, performance can be improved by up to 1.5x with serial fetching and 3x with round robin as it alleviates data dependencies between blocks and reduces the impact of the network on chip.
- An exploration of different configurations for a block based D-VTAGE value predictor with the new fetching scheme, resulting in an average 1.33x speedup compared to the current system, and up to a 2.7x performance increase.

This Chapter demonstrates that there is potential for more hardware support to make dynamic multi-core processors more practical.

Chapter 8

Conclusion

This thesis has explored how static ahead of time reconfiguration, dynamic runtime adaptation and micro-architectural modifications can make dynamic multi-core processors (DMP) more practical. Chapter 5 showed that a multi-threaded application can automatically be mapped ahead of time to a DMP, where different numbers of cores are composed for each thread. Chapter 6 demonstrated that at runtime the DMP can automatically change the size of a composition to ensure that the system is maximising speed whilst being energy efficient. Both chapters extract features from the software and use machine learning techniques such as linear regression and k Nearest Neighbours to determine correct configurations. Using cross-validation, the two chapters show that the automatic reconfiguration can be used in new unseen situations. The chapters also covered how source-level code optimisations can help improve the performance of core composition when access to the compiler is not available. Finally, Chapter 7 explored how modifying the fetching scheme in core composition and adding value prediction can improve the performance of large core compositions.

This chapter summarises the contributions of the thesis, followed by a critical analysis of chapters 5, 6 and 7. The final section covers future work in core composition.

8.1 Contributions

This section summarises the contributions of this thesis, specifically the work conducted in chapters 5, 6 and 7.

8.1.1 Static ahead of time thread and core partitioning

Chapter 5 tackled mapping streaming applications to a DMP. This process involved determining the best number of threads for an application and how many cores need to be composed per thread. The chapter first presented a design space exploration analysis of a set of streaming applications where 1500 different thread to core composition mappings were used. A compiler optimisation was also explored, loop unrolling, and showed that core composition is sensitive to block size: the larger the blocks the more likely core composition is going to more useful. Overall the design space exploration underlined that in order to get the best performance, a mix of multi-threading and core composition is required and leads to an average speedup of 3x compared to a single core single thread.

This was followed by the presentation of two models that can determine the number of threads that lead to the best performance for a given application and the number of cores per thread. The thread model used k Nearest Neighbour to classify a program based on its structure. The core composition model uses linear regression to determine how large a composition must be by analysing the average size of an unconditional block of operations found in the thread. Using leave one out cross-validation, automated model leads to DMP configurations that are within 16% of the best from the total exploration space.

This proved that a machine learning model can be used instead of hand-crafted heuristics to determine good configurations ahead of time by only analysing static features of an application. This means that if obtaining the fastest execution time is important, this thesis showed that this can be achieved automatically for DMPs without having to search the exhaustive space.

8.1.2 Dynamic runtime adaptation for efficient execution

Chapter 6 first covered how branch prediction and synchronisation costs affect the performance of different core composition sizes. It confirmed the early analysis of Chapter 5 that large EDGE blocks are critical to the efficient use of large core compositions: it reduces the branch prediction accuracy requirements and reduces the cost of synchronising cores. Then, an in depth comparison of dynamic and static ahead of time core compositions was conducted on a set of vision benchmarks. This study showed that whilst dynamic core compositions do not outperform static ahead of time compositions in terms of speed, dynamically changing the size of a composition can help

reduce energy consumption. By allowing the DMP to switch between compositions of sizes 1, 2, 4, 8 and 16 cores, dynamic adaptation can reduce energy consumption by 42% on average compared to a static ahead of time configuration.

The chapter then studied how the latency caused by switching the size of the composition can affect the energy savings. It was determined that, as phases are long, on average 100k cycles, the reconfiguration penalty can be between 100 to 1000 cycles without affecting savings. Finally, a simple linear regression was used to determine when to switch the size of a core composition at runtime. The model analysed the instruction mix of the blocks being executed to determine if the current composition was adequate. Using this automated adaptation scheme led to an average energy saving of 36% which was close to the best possible results.

This chapter underlined that a DMP can dynamically and automatically be reconfigured to reduce energy consumption at runtime without the need of profiling information for each application. This allows programmers to be able to immediately benefit from the DMP's flexibility without having to concern themselves with the added work of profiling their programs.

8.1.3 Adapting hardware to improve core composition performance

Chapter 7 proposed two hardware additions to the DMP in order to improve performance of core composition. These features aim to reduce data-dependencies between cores in a composition and also increase the percentage of time each core in a large composition was active. First, it analysed how the current fetching mechanism was a major cause of inefficiency for large core compositions when blocks are small. As the current fetching mechanism focuses on filling the instruction window of a single core, many cores in the composition are left idle. The chapter suggests a decentralised round-robin fetch scheme where cores fetch out of order and dispatch in order.

Second, the use of value prediction was motivated to reduce the penalty incurred by inter-block data dependencies. The chapter suggests that a block based computational value predictor be used as it allows multiple predictions to be generated quickly. This is followed by a performance analysis using a perfect value predictor with and without the round robin fetch scheme on the same benchmarks used in Chapter 6. The analysis shows that without value prediction, the round robin fetch scheme cannot improve performance due to the data dependencies found between blocks. However, when both value prediction and the round robin fetching scheme are used, this can improve the

performance of a 16 core composition by 1.8x on average. Finally, a block-based Differential VTAGE (D-VTAGE) value predictor was implemented and its performance analysed. Overall, using current state of the art value prediction with the round robin fetching scheme resulted in an average speedup of 1.33x, and could provide a speedup of up to 2.7x.

The results motivate more work in value prediction for core composition as it is an effective way to improve performance. It also demonstrates that more research must be conducted on how core composition should behave, as the simplest mechanisms may not be the most effective ones.

8.2 Critical Analysis

8.2.1 Simulation

The experiments done in this thesis use a cycle-accurate simulator for the EDGE architecture. This is due to the lack of any existing processor that uses the EDGE architecture and also has core composition. In fact, there exist no physical processor that implement core composition, thus simulation is the only current method of evaluating it. This is also an issue for value prediction, as no processor manufacturer has yet to implement one.

Whilst simulation can provide a good overview of how can be affected by different configurations of the processor, runtime adaptation, and the effect of value prediction, certain inaccuracies may arise from the implementation of the model. Furthermore, energy or power consumption can only be estimated through the use of a model. A Register-transfer level (RTL) implementation of the EDGE processor that features core composition and value prediction would allow for better precision.

8.2.2 Processor configuration

The design exploration and results produced in this thesis all used the same core configuration. Naturally, this means that some of the observations are specific to the processor that was used during the experiments. The thesis does provide a solid methodology for creating models that can predict a good configuration for any dynamic multi-core processor but it cannot provide absolute truths. For example, if cores can only fetch a single block at any given time, then composing cores would most likely improve performance at a faster rate, since the performance of a single core would be lower. This

means that the analysis of what loop optimisations make software run faster on compositions is potentially tied to the configuration of the cores. Thus, whilst the results presented throughout this thesis demonstrate that DMPs can be made more practical, it does not provide insights on when core composition should be a feature of a processor.

8.2.3 Compiler

In chapters 5 and 6, one of the factors that limits performance in core composition is block size. The chapters explored source-level transformations to improve the size of blocks. However code transformations at the compiler level could potentially increase block sizes for some programs that could not be improved manually. This could potentially affect the amount of instruction level parallelism found in some of the benchmarks explored throughout this thesis, which, naturally could affect energy savings when considering dynamic reconfiguration.

Furthermore hyperblock formation is currently limited to the merging of two blocks, which once again limits the potential size of a block. Hyperblock formation also does not take into account any profiling information which could potentially impact the formation of blocks. For example, the merging of two blocks that form an if/else statement is only useful if both conditions happen fairly regularly. If this is not the case and one of the statements takes most of the time then the block's size is only artificially increased, as some of the predicated instructions will never fire.

8.3 Future Work

Source level transformations were shown to help improve the performance of core composition, however they do not encompass all possible optimisations that can be applied to the code. Exploring different compiler level optimisations could potentially help make core compositions even more effective. For example, loops that feature a high amount of control flow currently cannot be unrolled as that still generates small blocks, as discussed in chapters 5 and 6. Employing optimisation techniques such as modulo scheduling can potentially help improve instruction level parallelism for such loops. As for hyperblock formation, using execution traces to determine which blocks should be merged together can help increase the usefulness of hyperblocks.

On the hardware side, it is important to evaluate when core composition should be implemented in a processor. Currently, the focus is on demonstrating that compos-

ing cores can lead to performance improvements, however there is no research being conducted on the types of cores that can benefit the most from composition. Exploring how core composition can speedup the performance of processors with different core configurations can help shed light as to when core composition should be implemented. This should be done by conducting a micro-architectural design space exploration where structures such as load-store queues, number of lanes a core has and cache sizes are modified.

Finally, pairing core composition with speculative parallel execution could provide an interesting avenue of research. For example, programs that feature irregular paralellism have phases that alternate between embarrassingly parallel work, and highly serialised work. A processor could begin by attempting to extract as much parallelism as possible via speculative parallelism, and adapt itself via core composition if the workload does not feature any parallelism. This could help improve the performance of graph algorithms that often feature phases of parallel and serial work.

Appendix A

Static ahead of time thread and core partitioning

This section shows the performance distribution graphs for all the benchmarks explored in Chapter 5. The curves represent the density distribution for different core compositions as a function of the number of threads. The performance (X-axis) is measured by comparing a specific design point to that of the single core single threaded execution. The reason the single-core single-thread is used as a baseline is due to the fact that it represents an ”unmodified” system. The right hand side Y-axis represents the number of threads present in the current version of the benchmark whilst the left Y-Axis represents the density normalized by the total number of points in the design space. For each of the threaded versions the benchmark runs using 100 different core-compositions. The density curve for thread 15 is a single point as there exists only a single composition, so a line is drawn to represent where that point lies.

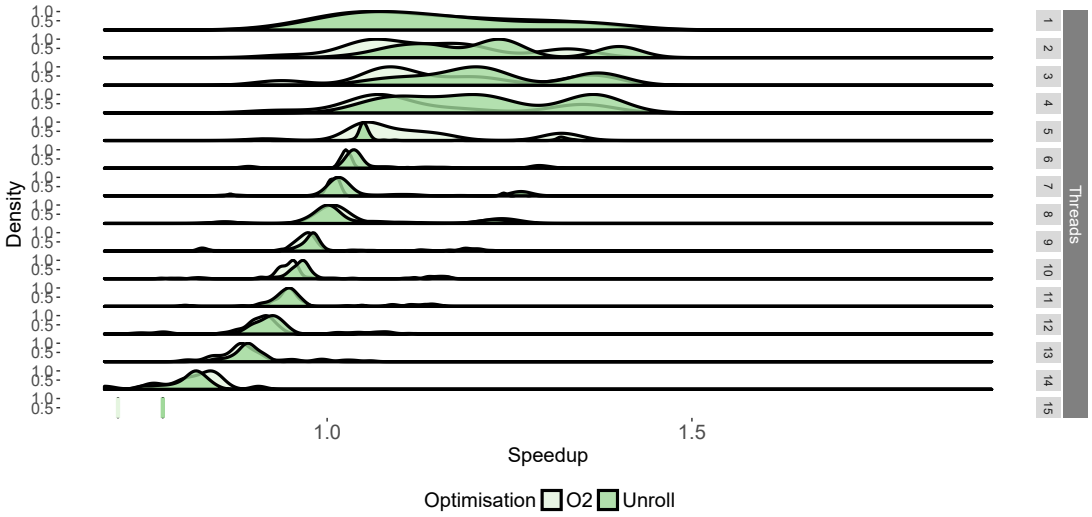


Figure A.1: Audio

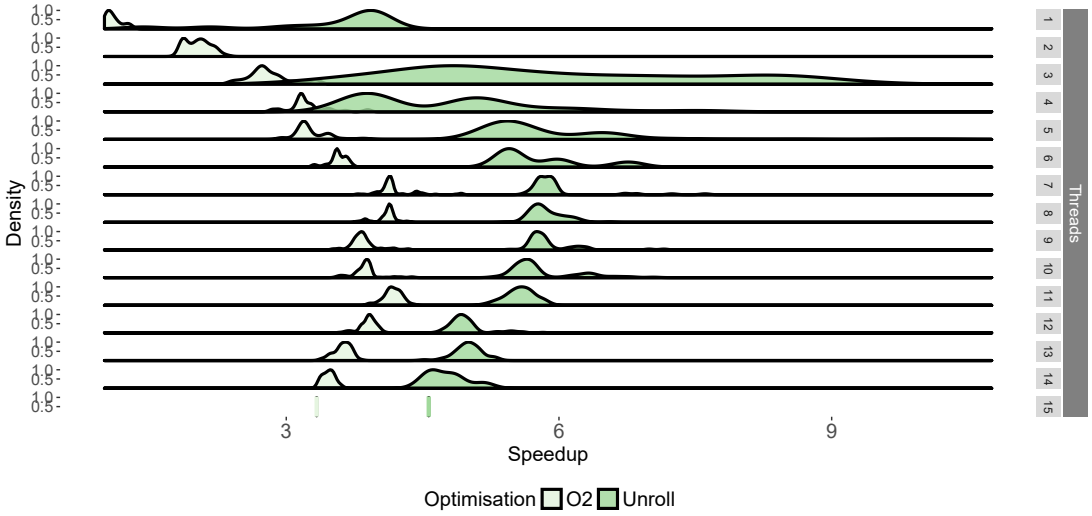


Figure A.2: Beam

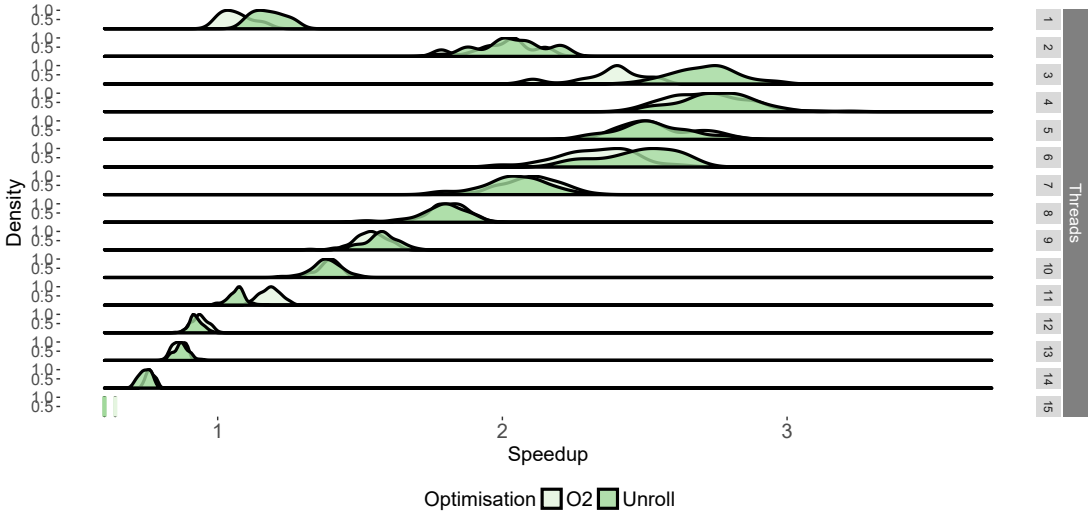


Figure A.3: Bitonic

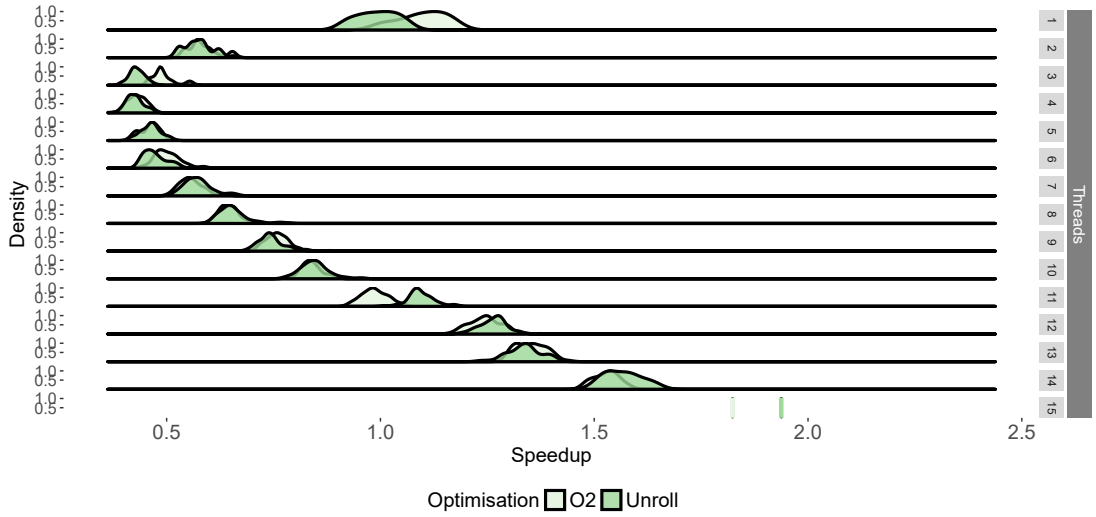


Figure A.4: Bubble

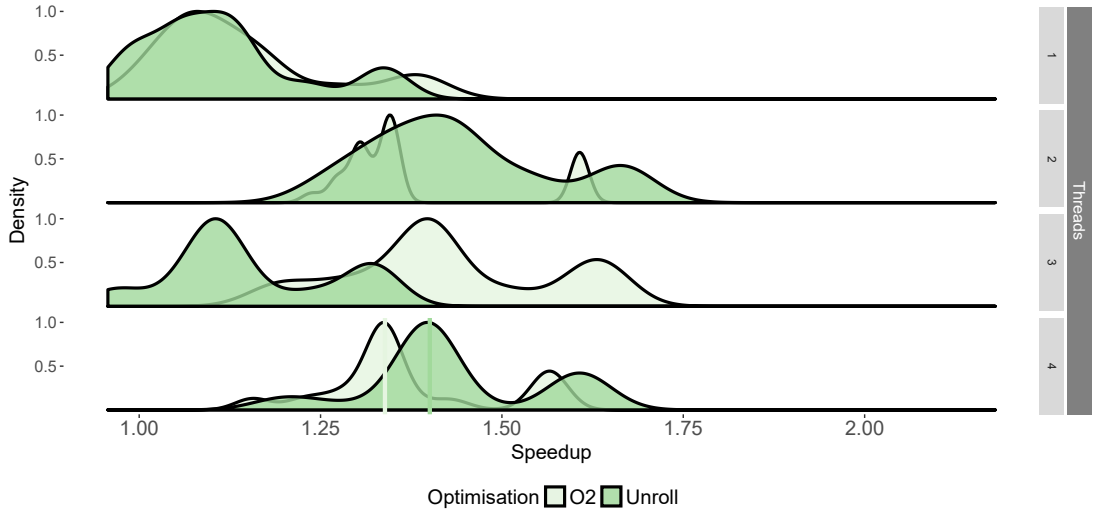


Figure A.5: CFAR

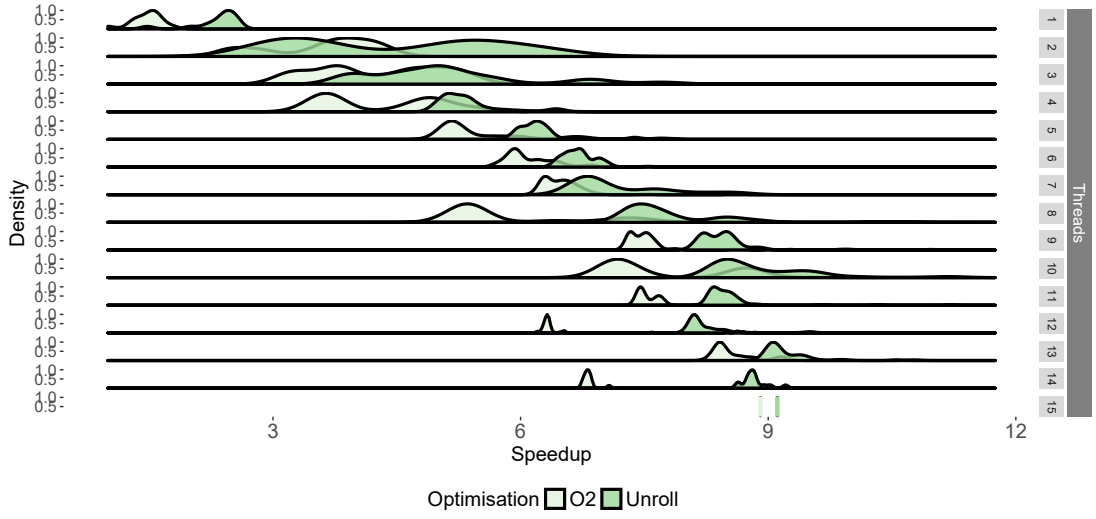


Figure A.6: Channel

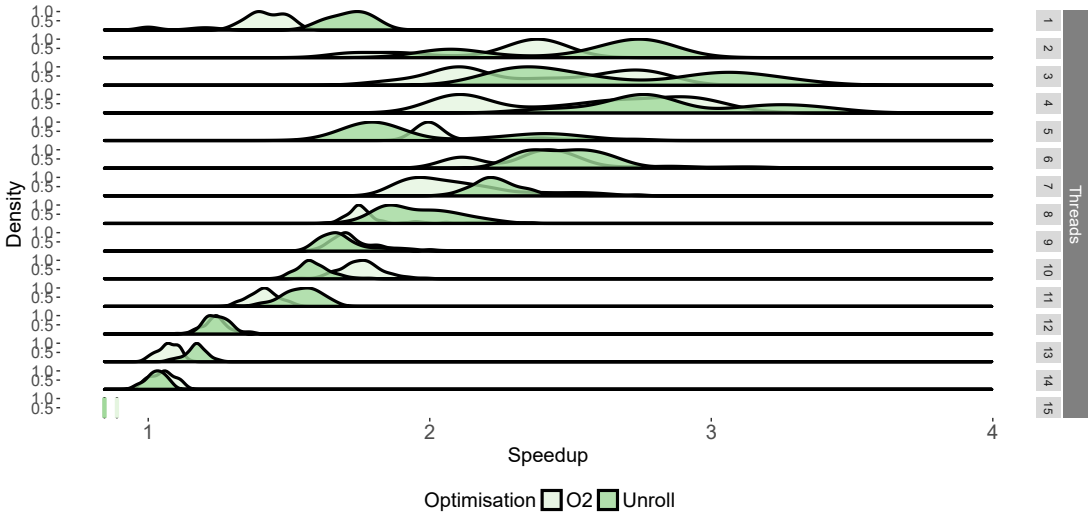


Figure A.7: FFT

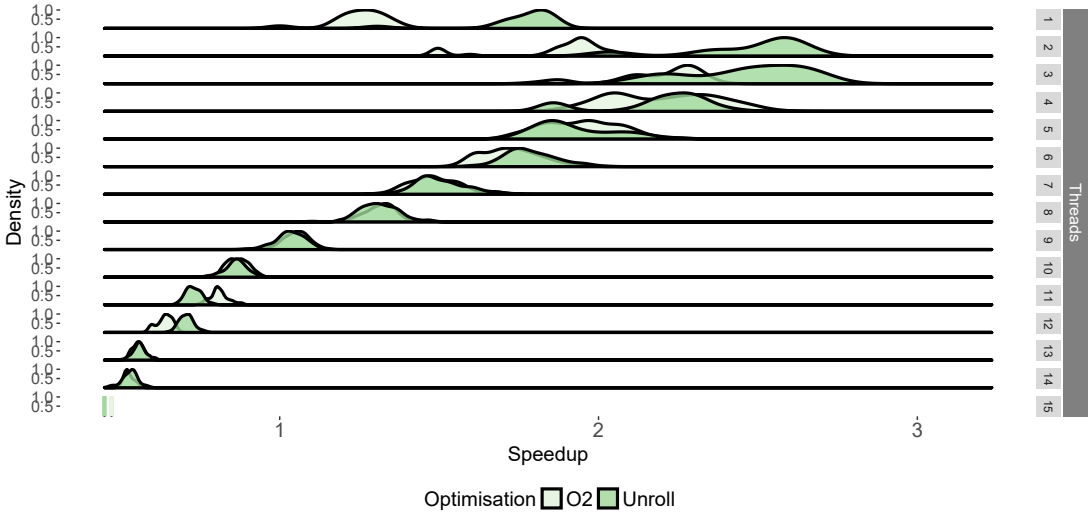


Figure A.8: FFT3

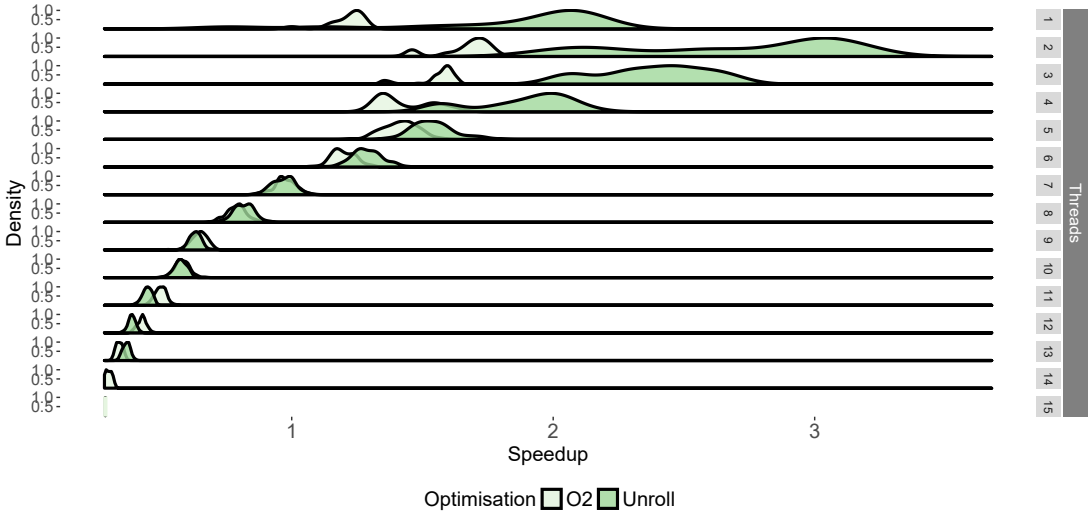


Figure A.9: FFT6

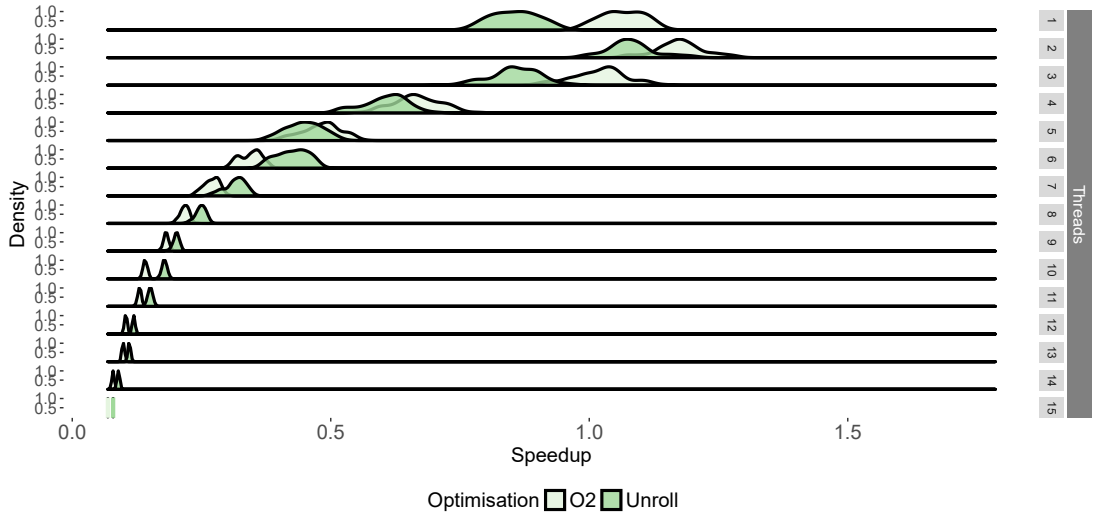


Figure A.10: FIR

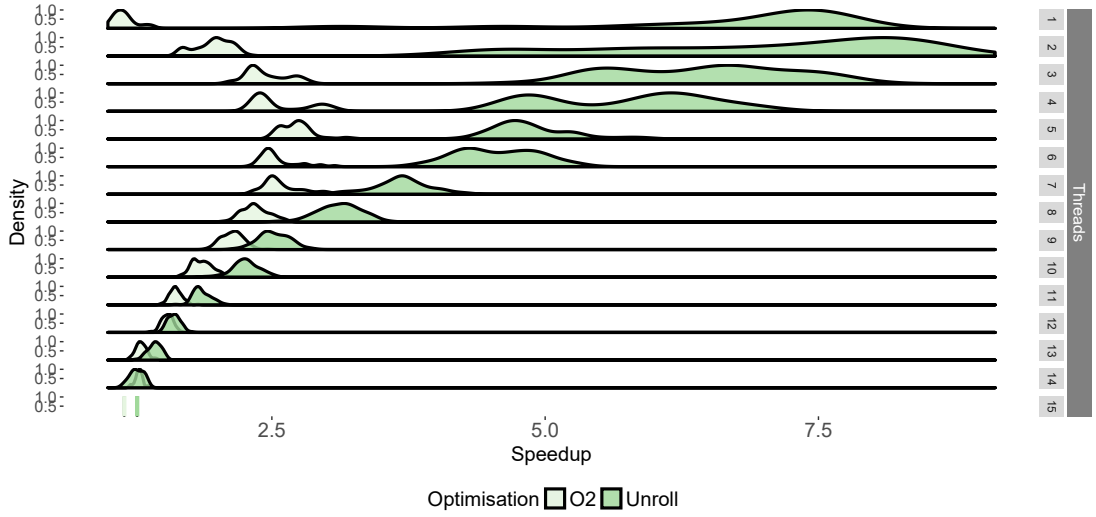


Figure A.11: FMRadio

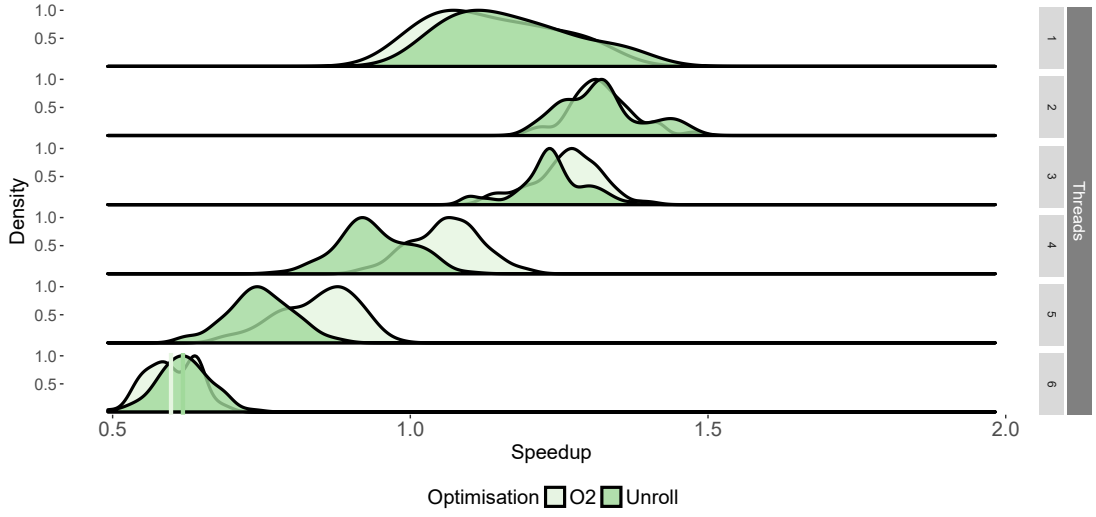


Figure A.12: Insertion

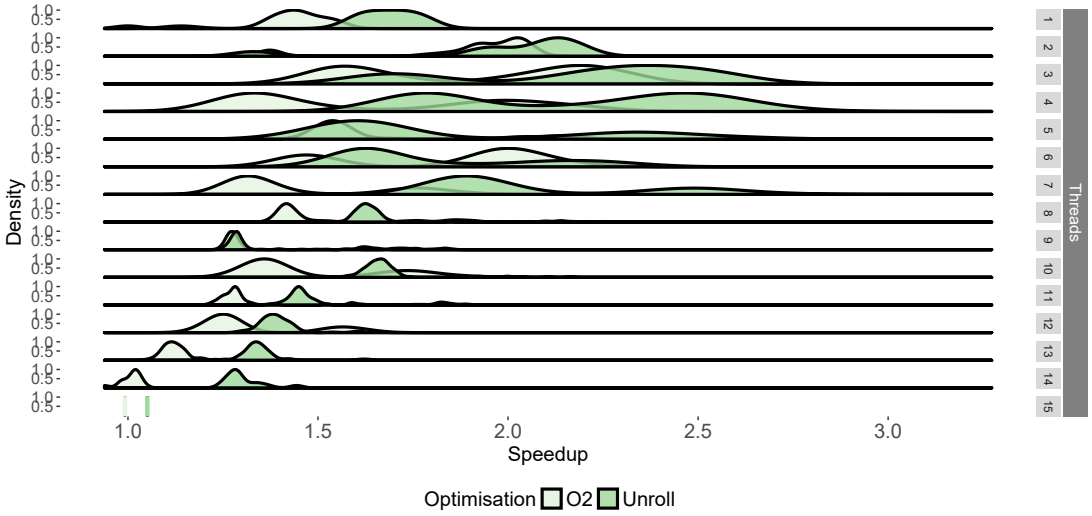


Figure A.13: Matmul

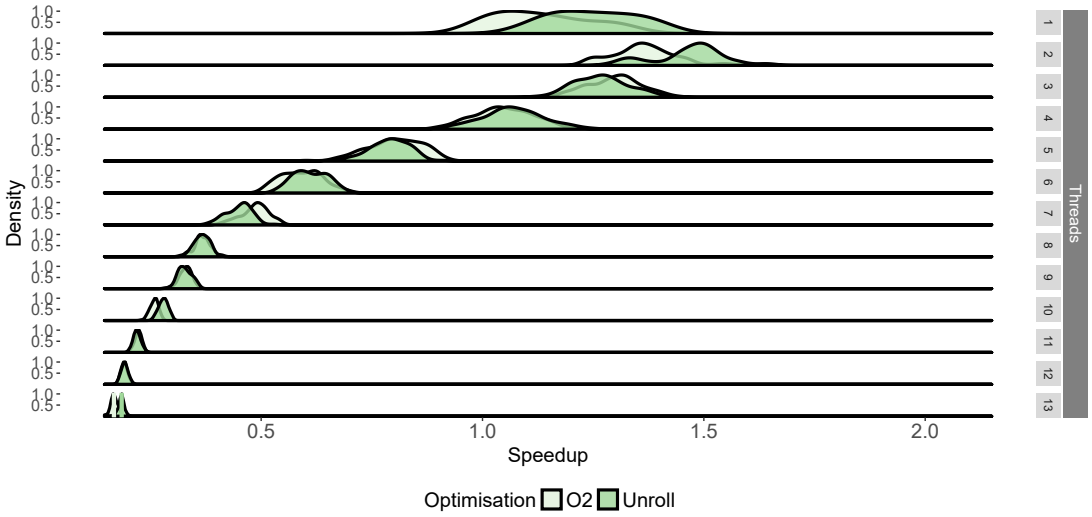


Figure A.14: Radix

Appendix B

Dynamic runtime adaptation for efficient execution

B.1 Code listings

```
1  int ndims = 2;
2  for(i = 0 ; i < nel ; ++i) {
3    idx_t index = pairs_pt [i].index ;
4    val_t value = pairs_pt [i].value ;
5    rindex = index ;
6    forest_pt [index] .parent   = index ;
7    forest_pt [index] .shortcut = index ;
8    forest_pt [index] .area     = 1 ;
9    idx_t temp = index ;
10   for(k = ndims-1 ; k >=0 ; --k) {
11     sref(nsubs_pt,k) = -1 ;
12     sref(subs_pt,k) = temp / sref(strides_pt,k) ;
13     temp          = temp % sref(strides_pt,k) ;
14   }
15   while(1) {
16     int good = 1 ;
17     idx_t nindex = 0 ;
18     for(k = 0 ; k < ndims && good ; ++k) {
19       int temp = sref(nsubs_pt,k) + sref(subs_pt,k) ;
20       good &= 0 <= temp && temp < sref(dims,k) ;
```

```
21        nindex += temp * sref(strides_pt,k) ;
22      }
23      if(good && nindex != index && forest_pt[nindex].parent ↩
           != node_is_void ) {
24        idx_t nrindex = 0, nvisited ;
25        val_t nrvalue = 0 ;
26        nvisited = 0 ;
27        while( forest_pt[rindex].shortcut != rindex ) {
28          sref(visited_pt,nvisited++) = rindex ;
29          rindex = forest_pt[rindex].shortcut ;
30        }
31        while( nvisited-- ) {
32          forest_pt [ sref(visited_pt,nvisited) ] .shortcut = ↩
               rindex ;
33        }
34        nrindex  = nindex ;
35        nvisited = 0 ;
36        while( forest_pt[nrindex].shortcut != nrindex ) {
37          sref(visited_pt, nvisited++) = nrindex ;
38          nrindex = forest_pt[nrindex].shortcut ;
39        }
40        while( nvisited-- ) {
41          forest_pt [ sref(visited_pt,nvisited) ] .shortcut = ↩
               nrindex ;
42        }
43        if( rindex != nrindex ) {
44          nrvalue = asubsref(I_pt,nrindex) ;
45          if( nrvalue == value) {
46            forest_pt[rindex] .parent   = nrindex ;
47            forest_pt[rindex] .shortcut = nrindex ;
48            forest_pt[nrindex].area    += forest_pt[rindex].↩
                 area ;
49            sref(joins_pt,njoins++) = rindex ;
50          }
51          else {
52            forest_pt[nrindex] .parent   = rindex ;
53            forest_pt[nrindex] .shortcut = rindex ;
```

```
54            forest_pt[rindex]  .area    += forest_pt[nrindex].↩
                 area ;
55            if( nrvalue != value ) {
56              forest_pt[nrindex].region = ner ;
57              regions_pt [ner] .index      = nrindex ;
58              regions_pt [ner] .parent     = ner ;
59              regions_pt [ner] .value      = nrvalue ;
60              regions_pt [ner] .area       = forest_pt [↩
                   nrindex].area ;
61              regions_pt [ner] .area_top   = nel ;
62              regions_pt [ner] .area_bot   = 0 ;
63              ++ner ;
64            }
65            sref(joins_pt,njoins++) = nrindex ;
66          }
67        }
68      }
69      k = 0 ;
70      sref(nsubs_pt,k) = sref(nsubs_pt,k) + 1;
71      while( sref(nsubs_pt, k) > 1) {
72        sref(nsubs_pt,k++) = -1 ;
73        if(k == ndims) goto done_all_neighbors ;
74        sref(nsubs_pt,k) = sref(nsubs_pt,k) + 1;
75      }
76    }
77    done_all_neighbors : ;
78  }
```

Listing B.1: MSER Compute extremal regions tree source code

```
I2D* fSortIndices(F2D* input, int dim)
{
    int rows, cols;
    F2D *in;
    int i, j, k;
    I2D *ind;

    rows = input->height;
    cols = input->width;
    in = fDeepCopy(input);
    ind = iMallocHandle(rows,cols);

    for(i=0; i<cols; i++)
        for(j=0; j<rows; j++)
            subsref(ind,j,i) = 0;

    for(k=0; k<rows; k++) {
        for(i=0; i<cols; i++) {
            float localMax = subsref(in,k,i);
            int localIndex = i;
            subsref(ind,k,i) = i;
            for(j=0; j<cols; j++) {
                if(localMax < subsref(in,k,j)) {
                    subsref(ind,k,i) = j;
                    localMax = subsref(in,k,j);
                    localIndex = j;
                }
            }
            subsref(in,k,localIndex) = 0;
        }
    }
    fFreeHandle(in);
    return ind;
}
```

```
35
36 int find(universe* u, int x) {
37     int y=x;
38     while (y != u->elts[y].p)
39         y = u->elts[y].p;
40     u->elts[x].p = y;
41     return y;
42 }
43
44 void join(universe* u, int x, int y){
45     if (u->elts[x].rank > u->elts[y].rank) {
46         u->elts[y].p = x;
47         u->elts[x].size += u->elts[y].size;
48     }
49     else {
50         u->elts[x].p = y;
51         u->elts[y].size += u->elts[x].size;
52         if (u->elts[x].rank == u->elts[y].rank)
53             u->elts[y].rank++;
54     }
55     u->num--;
56     return ;
57 }
58
59 universe *segment_graph(int num_vertices, int num_edges, ↩
    edge *edges, float c) {
60     float* threshold = alloca(sizeof(int) * num_vertices);
61     int i, a, b, j, k;
62     universe *u;
63     F2D *edgeWeig!htbs;
64     I2D *indices;
65     edgeWeig!htbs = fMallocHandle(1,num_edges);
66
67     for(i=0; i<num_edges; i++)
68         asubsref(edgeWeig!htbs,i) = edges[i].w;
69
70     indices = fSortIndices(edgeWeig!htbs,1);
```

```
71      u = (universe*)malloc(sizeof(universe));
72      u->elts = (uni_elt*)malloc(sizeof(uni_elt)*num_vertices)↩
            ;
73      u->num = num_vertices;
74
75      for(i=0; i<num_vertices; i++)     {
76          u->elts[i].rank = 0;
77          u->elts[i].size = 1;
78          u->elts[i].p = i;
79      }
80
81      for (i = 0; i < num_vertices; i++)
82          arrayref(threshold,i) = THRESHOLD(1,c);
83
84      for (i = 0; i < num_edges; i++) {
85          edge *pedge = &edges[ asubsref(indices,i) ];
86      a = find(u, pedge->a);
87          b = find(u, pedge->b);
88          if (a != b) {
89              if ((pedge->w <= arrayref(threshold,a)) && (↩
                    pedge->w <= arrayref(threshold,b))) {
90                join(u, a, b);
91                a = find(u, a);
92                arrayref(threshold,a) = pedge->w + THRESHOLD(u↩
                      ->elts[a].size, c);
93              }
94          }
95      }
96    return u;
97  }
```

Listing B.2: Multi_NCut main phase of segmenting image

Bibliography

[Abey 17] M. Abeydeera, S. Subramanian, M. C. Jeffrey, J. Emer, and D. Sanchez. "SAM: Optimizing Multithreaded Cores for Speculative Parallelism". In: *PACT*, pp. 64–78, 2017.

[Adil 16] A. Adileh, S. Eyerman, A. Jaleel, and L. Eeckhout. "Maximizing Heterogeneous Processor Performance Under Power Constraints". *TACO*, Vol. 13, 2016.

[Ains 16] S. Ainsworth and T. M. Jones. "Graph Prefetching Using Data Structure Knowledge". In: *ISC*, pp. 39:1–39:11, 2016.

[Ains 17] S. Ainsworth and T. M. Jones. "Software Prefetching for Indirect Memory Accesses". In: *CGO*, pp. 305–317, 2017.

[Amda 67] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *SJCC*, pp. 483–485, 1967.

[Anan 18a] Anandtech. "The A12 Vortex CPU microarchitecture". https://www.anandtech.com/show/13392/the-iphone-xs-xs-max-review-unveiling-the-silicon-secrets/3, 2018. Accessed: 2018-11-20.

[Anan 18b] Anandtech. "The Samsung Exynos M3 6 Wide Decode With 50 https://www.anandtech.com/show/12361/samsung-exynos-m3-architecture, 2018. Accessed: 2018-11-20.

[ARM 13] R. ARM. "big. LITTLE Technology: The Future of Mobile". https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf, 2013.

[Asan 06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.* "The landscape of parallel computing research: A view from berkeley". Tech. Rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[Auer 12] J. Auerbach, D. Bacon, I. Burcea, P. Cheng, S. Fink, R. Rabbah, and S. Shukla. "A compiler and runtime for heterogeneous computing". In: *DAC*, pp. 271–276, 2012.

[Baue 08] L. Bauer, M. Shafique, S. Kreutz, and J. Henkel. "Run-time System for an Extensible Embedded Processor with Dynamic Instruction Set". In: *DATE*, 2008.

[Bell 08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect". In: *ISSCC*, pp. 88–598, 2008.

[Bosb 14] J. Bosboom, S. Rajadurai, W. Wong, and S. Amarasinghe. "StreamJIT: A Commensal Compiler for High-performance Stream Programming". *SIGPLAN Not.*, Vol. 49, 2014.

[Buck 04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. "Brook for GPUs: Stream Computing on Graphics Hardware". In: *ACM SIGGRAPH 2004*, pp. 777–786, 2004.

[Burg 04] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, X. Chen, R. Desikan, S. Drolia, J. Gibson, M. S. Govindan, P. Gratz, H. Hanson, C. Kim, S. K. Kushwaha, H. Liu, R. Nagarajan, N. Ranganathan, R. Reeber, K. Sankaralingam, S. Sethumadhavan, P. Sivakumar, and A. Smith. "Scaling to the End of Silicon with EDGE Architectures". *Computer*, Vol. 37, 2004.

[Burg 18] D. Burger, A. Smith, and G. Wright. "ISCA 2018 Keynote Speakers". `https://iscaconf.org/isca2018/keynotes.html`, 2018. Accessed: 2018-11-20.

[Burg 97] D. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0". *ACM SIGARCH computer architecture news*, Vol. 25, 1997.

[Carp 09] P. M. Carpenter, A. Ramirez, and E. Ayguade. "Mapping Stream Programs Onto Heterogeneous Multiprocessor Systems". In: *CASES*, pp. 57–66, 2009.

[Chen 05] J. Chen, M. I. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand. "A Reconfigurable Architecture for Load-balanced Rendering". In: *HWWS*, pp. 71–80, 2005.

[Chry 98] G. Z. Chrysos and J. S. Emer. "Memory Dependence Prediction Using Store Sets". *SIGARCH Comput. Archit. News*, Vol. 26, 1998.

[Cumm 17a] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. "End-to-End Deep Learning of Optimization Heuristics". In: *PACT*, pp. 219–232, 2017.

[Cumm 17b] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. "Synthesizing benchmarks for predictive modeling". In: *CGO*, pp. 86–99, 2017.

[Dagu 98] L. Dagum and R. Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". *IEEE Comput. Sci. Eng.*, Vol. 5, 1998.

[DeVu 12] M. DeVuyst, A. Venkat, and D. M. Tullsen. "Execution Migration in a heterogeneous-ISA Chip Multiprocessor". *SIGPLAN Not.*, Vol. 47, 2012.

[Duba 12] C. Dubach, T. M. Jones, and M. F. P. O'Boyle. "Exploring and Predicting the Effects of Microarchitectural Parameters and Compiler Optimizations on Performance and Energy". *TECS*, Vol. 11S, 2012.

[Duba 13] C. Dubach, J. T. M., and B. E. V. "Dynamic Microarchitectural Adaptation using Machine Learning". *TACO*, Vol. 10, 2013.

[Edin 16] U. of Edinburgh. "Edinburgh Compute and Data Facility Web Site". 1 August 2007, accessed 4th of April. 2016. `www.ecdf.ed.ac.uk`.

[Euro 19a] EuroLab4HPC. "Aaron Smith; Microsoft & Univeristy of Edinburgh - Is it Time for RISC and CISC to Die ?". `https://www.youtube.com/watch?v=VNGuTE6OUGc&t=1413s`, 2019. Accessed: 2019-03-09.

[Euro 19b] EuroLab4HPC. "Eurolab4HPC Industry Day Co-organized with Multicore Day 2018". https://www.eurolab4hpc.eu/communication/events/24/eurolab4hpc-industry-day-co-organized-with-multicore-day-2018/, 2019. Accessed: 2019-03-09.

[Ever 01] L. Everitt, Landau. *Cluster Analysis*. 2001.

[Eyer 10] S. Eyerman and L. Eeckhout. "Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design". *SIGARCH Comput. Archit. News*, Vol. 38, 2010.

[Fall 14] C. Fallin, C. Wilkerson, and O. Mutlu. "The Heterogeneous Block Architecture". In: *ICCD*, pp. 386–393, 2014.

[Farh 12] S. M. Farhad, Y. Ko, B. Burgstaller, and B. Scholz. "Profile-guided Deployment of Stream Programs on Multicores". pp. 79–88, 2012.

[Gabb 98] F. Gabbay and A. Mendelson. "Using Value Prediction to Increase the Power of Speculative Execution Hardware". *ACM Trans. Comput. Syst.*, Vol. 16, 1998.

[Goem 01] B. Goeman, H. Vandierendonck, and K. de Bosschere. "Differential FCM: increasing value prediction accuracy by improving table usage efficiency". In: *HPCA*, pp. 207–216, 2001.

[Gord 02] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. "A Stream Compiler for Communication-exposed Architectures". *SIGARCH Comput. Archit. News*, Vol. 30, 2002.

[Gray 18a] J. Gray and A. Smith. "Towards an Area-Efficient Implementation of a High ILP EDGE Soft Processor". *CoRR*, Vol. abs/1803.06617, 2018.

[Gray 18b] J. Gray. "ISCA 2018 E2 Keynote Summary". https://twitter.com/jangray/status/1004874394957578242, 2018. Accessed: 2018-11-20.

[Gula 08] D. P. Gulati, C. Kim, S. Sethumadhavan, S. W. Keckler, and D. Burger. "Multitasking Workload Scheduling on Flexible Core Chip Multiprocessors". *SIGARCH Comput. Archit. News*, Vol. 36, 2008.

[Gupt 17] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras. "DyPO: Dynamic Pareto-Optimal Configuration Selection for Heterogeneous MpSoCs". *TECS*, Vol. 16, 2017.

[Guth 01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE WWC*, pp. 3–14, 2001.

[Haqu 17] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley. "Exploiting Heterogeneity for Tail Latency and Energy Efficiency". In: *MICRO*, pp. 625–638, 2017.

[Henn 00] J. L. Henning. "SPEC CPU2000: Measuring CPU Performance in the New Millennium". *Computer*, Vol. 33, 2000.

[Herb 07] S. Herbert and D. Marculescu. "Analysis of Dynamic Voltage/Frequency Scaling in Chip-multiprocessors". In: *ISPLED*, pp. 38–43, 2007.

[Hert 11] B. Hertzberg and O. K. "Runtime automatic speculative parallelization". In: *CGO*, pp. 64–73, 2011.

[Homa 12] H. Homayoun, V. Kontorinis, A. Shayan, T. Lin, and D. M. Tullsen. "Dynamically Heterogeneous Cores Through 3D Resource Pooling". In: *HPCA*, pp. 1–12, 2012.

[Inte 16] Intel. "Intel 64 and IA-32 Architectures Optimization Reference Manual". `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, 2016.

[Ipek 07] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors". In: *ISCA*, pp. 186–197, 2007.

[Jafr 13] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar. "Wait-n-GoTM: Improving HTM Performance by Serializing Cyclic Dependencies". In: *ASPLOS*, pp. 521–534, 2013.

[Jeff 12] B. Jeff. “Big. LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration”. In: *DAC*, pp. 1143–1146, 2012.

[Jeff 16] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. “Unlocking Ordered Parallelism with the Swarm Architecture”. *IEEE Micro*, Vol. 36, 2016.

[Jose 06] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. “Construction and use of linear regression models for processor performance analysis”. In: *The Twelfth HPCA, 2006.*, pp. 99–108, 2006.

[Kanu 02] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. “An Efficient k-Means Clustering Algorithm: Analysis and Implementation”. *TPAMI*, Vol. 24, 2002.

[Khub 12] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. “MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP”. In: *MICRO*, 2012.

[Kim 07] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. “Composable Lightweight Processors”. In: *Proceedings of the 40th Annual IEEE/ACM MICRO*, pp. 381–394, 2007.

[Kirk 83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by simulated annealing”. *science*, Vol. 220, 1983.

[Koch 13] T. J. Edler von Koch and B. Franke. “Limits of Region-based Dynamic Binary Parallelization”. In: *VEE*, pp. 13–22, 2013.

[Kudl 08] M. Kudlur and S. Mahlke. “Orchestrating the Execution of Stream Programs on Multicore Platforms”. *SIGPLAN Not.*, Vol. 43, 2008.

[Lee 06] B. C. Lee and D. M. Brooks. “Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction”. *SIGPLAN Not.*, Vol. 41, 2006.

[Lee 12] J. Lee, H. Kim, and R. Vuduc. “When Prefetching Works, When It Doesn’t, and Why”. *TACO*, Vol. 9, 2012.

[Lee 97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems". In: *Proceedings of 30th Annual MICRO*, pp. 330–335, 1997.

[Li 09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures". In: *MICRO*, pp. 469–480, 2009.

[Lipa 96] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. "Value Locality and Load Value Prediction". *SIGPLAN Not.*, Vol. 31, 1996.

[Lloy 82] S. Lloyd. "Least squares quantization in PCM". *IEEE Transactions on Information Theory*, Vol. 28, 1982.

[Luke 12] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. "Composite Cores: Pushing Heterogeneity Into a Core". In: *Proceedings of the 2012 45th Annual IEEE/ACM MICRO*, pp. 317–328, 2012.

[Mico 16] P. J. Micolet, A. Smith, and C. Dubach. "A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors". In: *LCTES*, pp. 113–122, 2016.

[Mico 17] P. J. Micolet, A. Smith, and C. Dubach. "A Study of Dynamic Phase Adaptation Using a Dynamic Multicore Processor". *TECS*, Vol. 16, 2017.

[Migu 14] J. S. Miguel, M. Badr, and N. E. Jerger. "Load Value Approximation". In: *MICRO*, pp. 127–139, 2014.

[Mitt 16] S. Mittal. "A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors". *ACM Computing Surveys*, Vol. 48, 2016.

[N 06] R. N. and Z. C. "Probabilistic counter updates for predictor hysteresis and stratification". In: *HPCA*, pp. 110–120, 2006.

[Newt 08] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. "Design and Evaluation of a Compiler for Embedded Stream Programs". In: *LCTES*, pp. 131–140, 2008.

[Paga 13] S. Pagani and J. Chen. "Energy Efficient Task Partitioning Based on the Single Frequency Approximation Scheme". In: *RTSS*, pp. 308–318, 2013.

[Paga 17] S. Pagani, A. Pathania, M. Shafique, J. J. Chen, and J. Henkel. "Energy Efficiency for Clustered Heterogeneous Multicores". *TPDS*, Vol. 28, 2017.

[Pera 14] A. Perais and A. Seznec. "Practical data value speculation for future high-end processors". In: *HPCA*, pp. 428–439, 2014.

[Pera 15] A. Perais and A. Seznec. "BeBoP: A cost effective predictor infrastructure for superscalar value prediction". In: *HPCA*, pp. 13–25, 2015.

[Pere 03] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. "Using SimPoint for Accurate and Efficient Simulation". In: *Proceedings of the 2003 ACM SIGMETRICS*, pp. 318–319, 2003.

[Ping 11] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. "The Tao of Parallelism in Algorithms". In: *Proceedings of the 32Nd ACM SIGPLAN PLDI*, pp. 12–25, 2011.

[Poov 09] J. A. Poovey, T. M. Conlte, M. Levy, and S. Gal-On. "A Benchmark Characterization of the EEMBC Benchmark Suite". *IEEE Micro*, Vol. 29, 2009.

[Prab 11] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. "Commutative Set: A Language Extension for Implicit Parallel Programming". In: *Proceedings of the 32Nd ACM SIGPLAN PLDI*, pp. 1–11, 2011.

[Pric 12] M. Pricopi and T. Mitra. "Bahurupi: A Polymorphic Heterogeneous Multi-core Architecture". *TACO*, Vol. 8, 2012.

[Pric 14] M. Pricopi and T. Mitra. "Task Scheduling on Adaptive Multi-Core". *IEEE Transactions on Computer*, Vol. 63, 2014.

[Putn 11] A. Putnam, A. Smith, and D. Burger. "Dynamic Vectorization in the E2 Dynamic Multicore Architecture". *SIGARCH Comput. Archit. News*, Vol. 38, 2011.

[Rauc 95] L. Rauchwerger and D. Padua. “The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization”. In: *PLDI*, pp. 218–232, 1995.

[Regi 18] T. Register. “Now Microsoft ports Windows 10, Linux to home-grown CPU design”. `https://www.theregister.co.uk/2018/06/18/microsoft_e2_edge_windows_10/`, 2018. Accessed: 2018-11-20.

[Roba 11] B. Robatmili, S. Govindan, D. Burger, and S. W. Keckler. “Exploiting criticality to reduce bottlenecks in distributed uniprocessors”. In: *HPCA*, pp. 431–442, 2011.

[Rodr 14] R. Rodrigues, I. Koren, and S. Kundu. “Performance and Power Benefits of Sharing Execution Units between a High Performance Core and a Low Power Core”. In: *VLSI*, pp. 204–209, 2014.

[Sezn 11] A. Seznec. “A New Case for the TAGE Branch Predictor”. In: *MICRO*, pp. 117–127, 2011.

[Sezn 96] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. “Multiple-block Ahead Branch Predictors”. *SIGOPS Oper. Syst. Rev.*, Vol. 30, 1996.

[Shei 17] R. Sheikh, H. W. Cain, and R. Damodaran. “Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions Due to Conflicting Stores”. In: *MICRO*, pp. 423–435, 2017.

[Smit 06a] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. Burrill. “Compiling for EDGE Architectures”. In: *CGO*, 2006.

[Smit 06b] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. “Dataflow Predication”. In: *MICRO*, pp. 89–102, 2006.

[Ston 10] J. E. Stone, D. Gohara, and G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. *Computing in Science Engineering*, Vol. 12, 2010.

[Subr 17] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez. “Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism”. In: *ISCA*, pp. 587–599, 2017.

[T 18] W. T. “StreamIt Github Repository”. 2018. https://github.com/bthies/streamit.

[Tava 15] M. K. Tavana, M. H. Hajkazemi, D. Pathak, I. Savidis, and H. Homayoun. “ElasticCore: Enabling Dynamic Heterogeneity with Joint Core and Voltage/Frequency Scaling”. In: *DAC*, pp. 1–6, 2015.

[Thie 02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *CC*, pp. 179–196, 2002.

[Thie 10] W. Thies and S. Amarasinghe. “An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design”. In: *PACT*, pp. 365–376, 2010.

[Turl 14] J. Turley. “White Paper Introduction to Intel® Architecture”. 2014.

[Vega 13] A. Vega, A. Buyuktosunoglu, H. Hanson, P. Bose, and S. Ramani. “Crank it up or dial it down: Coordinated multiprocessor frequency and folding control”. In: *2013 46th Annual IEEE/ACM MICRO (MICRO)*, pp. 210–221, 2013.

[Venk 09] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor. “SD-VBS: The San Diego Vision Benchmark Suite”. In: *IISWC*, pp. 55–64, 2009.

[Venk 14] A. Venkat and D. M. Tullsen. “Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor”. In: *ISCA*, pp. 121–132, 2014.

[Venk 16] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen. “HIPStR: Heterogeneous-ISA Program State Relocation”. *SIGPLAN Not.*, Vol. 51, 2016.

[Vudu 03] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, 2003. AAI3121741.

[Wain 97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. “Baring it all to software: Raw machines”. *Computer*, Vol. 30, 1997.

[Wang 08] Z. Wang and M. O’Boyle. “Using Machine Learning to Partition Streaming Programs”. *TACO*, Vol. 10, 2008.

[Wata 10] Y. Watanabe, J. D. Davis, and D. A. Wood. “WiDGET: Wisconsin Decoupled Grid Execution Tiles”. In: *ISCA*, pp. 2–13, 2010.